LET OVER LAMBDA

LET OVER LAMBDA

50 years of Lisp

DOUG HOYTE

Let Over Lambda

50 Years of Lisp

Doug Hoyte

λ

Let Over Lambda
Edition 1.0

Copyright ©2008 by Doug Hoyte
All rights reserved

Code is copyright ©2002-2008 by Doug Hoyte
No rights reserved

ISBN: 978-1-4357-1275-1

http://letoverlambda.com
doug@hcsw.org

An HCSW and Hoytech production
Made with lisp

Contents

Contents i

1 Introduction 1
- 1.1 Macros . . . 1
- 1.2 U-Language . . . 7
- 1.3 The Lisp Utility . . . 12
- 1.4 License . . . 15
- 1.5 Thanks . . . 16

2 Closures 17
- 2.1 Closure-Oriented Programming . . . 17
- 2.2 Environments and Extent . . . 18
- 2.3 Lexical and Dynamic Scope . . . 20
- 2.4 Let It Be Lambda . . . 25
- 2.5 Let Over Lambda . . . 31
- 2.6 Lambda Over Let Over Lambda . . . 33
- 2.7 Let Over Lambda Over Let Over Lambda . . . 36

3 Macro Basics 39
- 3.1 Iterative Development . . . 39
- 3.2 Domain Specific Languages . . . 40
- 3.3 Control Structures . . . 44
- 3.4 Free Variables . . . 48
- 3.5 Unwanted Capture . . . 51
- 3.6 Once Only . . . 64
- 3.7 Duality of Syntax . . . 71

4 Read Macros **75**
4.1 Run-Time at Read-Time 75
4.2 Backquote 78
4.3 Reading Strings 83
4.4 CL-PPCRE 87
4.5 Cyclic Expressions 95
4.6 Reader Security 98

5 Programs That Program **107**
5.1 Lisp Is Not Functional 107
5.2 Top-Down Programming 111
5.3 Implicit Contexts 120
5.4 Code-Walking with Macrolet 126
5.5 Recursive Expansions 132
5.6 Recursive Solutions 140
5.7 Dlambda 147

6 Anaphoric Macros **153**
6.1 More Phors? 153
6.2 Sharp-Backquote 156
6.3 Alet and Finite State Machines 159
6.4 Indirection Chains 168
6.5 Hotpatching Closures 174
6.6 Sub-Lexical Scope 178
6.7 Pandoric Macros 187

7 Macro Efficiency Topics **207**
7.1 Lisp Is Fast 207
7.2 Macros Make Lisp Fast 209
7.3 Getting To Know Your Disassembler 225
7.4 Pointer Scope 232
7.5 Tlists and Cons Pools 243
7.6 Sorting Networks 254
7.7 Writing and Benchmarking Compilers 269

8 Lisp Moving Forth Moving Lisp **285**
8.1 Weird By Design 285
8.2 Cons Threaded Code 289
8.3 Duality of Syntax, Defined 296
8.4 Going Forth 304
8.5 Going Forther 315
8.6 Going Lisp 328

Appendices **340**

A Languages to Learn **341**
A.1 Road to Lisp 341
A.2 C and Perl 342
A.3 Lisp Incubators 342

B Languages to Avoid **345**
B.1 Opinion 345
B.2 Blub Central 345
B.3 Niche Blub 347

C Implementations **349**
C.1 CMUCL/SBCL 349
C.2 CLISP 350
C.3 Others 350

D Lisp Editors **353**
D.1 emacs 353
D.2 vi 354

References **357**

References by Author **367**

Index **372**

Chapter 1

Introduction

1.1 Macros

> Lisp's core occupies some kind of local optimum in the space of programming languages.
>
> —Modest words from John McCarthy, inventor of lisp

This is a book about programming *macros* in lisp. Unlike most programming books that only give them a cursory overview, this book is a series of tutorials and examples designed to get you programming sophisticated macros as efficiently and quickly as possible. Mastering macros is the final step to graduating from an intermediate lisp programmer to a lisp professional.

Macros are the single greatest advantage that lisp has as a programming language and the single greatest advantage of any programming language. With them you can do things that you simply cannot do in other languages. Because macros can be used to transform lisp into other programming languages and back, programmers who gain experience with them discover that all other languages are just skins on top of lisp. This is the *big deal.* Lisp is special because programming with it is actually programming at a higher level. Where most languages invent and enforce syntactic and semantic rules, lisp is general and malleable. With lisp, you make the rules.

Lisp has a richer, deeper history than all other programming languages. Some of the best and brightest computer scientists throughout our field's brief existence have worked very hard to make it the most powerful and general programming language ever. Lisp also enjoys a number of concise standards, multiple excellent open-source implementations, and more macro conveniences than any other programming language. This book uses only COMMON LISP[ANSI-CL][CLTL2] but many of the ideas are trivially portable to other lisps like Scheme[R5RS]. That said, hopefully this book will convince you that if you want to write macros, COMMON LISP is the lisp to use. While different types of lisp are excellent for other purposes, COMMON LISP is deservedly the tool of choice for the macro professional.

The designers of COMMON LISP have done an excellent job in designing a programming language *right*. Especially after considering implementation quality, COMMON LISP is, with surprisingly few reservations, by far the best programming environment available to current-day programmers. As a programmer you can almost always count on COMMON LISP to have done it the way it ought be done. While it's true the designers and implementors have done it the right way, some feel that they forgot to describe to us just why it's right. To many outsiders COMMON LISP just looks like an enormous collection of strange features, and so are turned off, moving to a more immediately gratifying language—doomed to never experience the true power of macros. Although it is not its primary purpose, this book can be used as a tour of many of the best features in the amazing language that is COMMON LISP. Most languages are designed to be easy to implement; COMMON LISP is designed to be powerful to program. I sincerely hope the creators of COMMON LISP appreciate this book as one of the most complete and accessible treatments of the language's advanced macro features, and also as an enjoyable drop in the ocean of topics that is the macro.

Macros have, not by accident, almost as much history as lisp itself, being invented in 1963 by Timothy Hart[MACRO-DEFINITIONS]. However, macros are still not used to the fullest possible extent by most lisp programmers and are not used at all by all other

programmers. This has always been a conundrum for advanced lispers. Since macros are so great, why doesn't everybody use them all the time? While it's true that the smartest, most determined programmers always end up at lisp macros, few start their programming careers there. Understanding why macros are so great requires understanding what lisp has that other languages don't. It requires an understanding of other, less powerful languages. Sadly, most programmers lose the will to learn after they have mastered a few other languages and never make it close to understanding what a macro is or how to take advantage of one. But the top percentile of programmers in any language are always forced to learn some sort of way to write programs that write programs: macros. Because it is the best language for writing macros, the smartest and most determined and most curious programmers always end up at lisp.

Although the top-percentile of programmers is necessarily a small number, as the overall programming population grows so does the number of top-percentile programmers. The programming world sees few examples of the power of macros and understands far fewer, but this is changing. Because of the productivity multiplication that can be achieved through macros, the *age of the macro* is coming, whether the world is ready or not. This book aims to be a base-line preparation for the inevitable future: a world of macros. Be prepared.

The conventional wisdom surrounding macros is to use them only when necessary because they can be difficult to understand, contain extremely subtle bugs, and limit you in possibly surprising ways if you think of everything as functions. These aren't defects in the lisp macro system itself but instead are traits of macro programming in general. As with any technology, the more powerful the tool, the more ways there are to misuse it. And, as far as programming constructs go, lisp macros are the most powerful tool.

An interesting parallel to learning macros in lisp is that of learning pointers in the C programming language. Most beginning C programmers are able to quickly pick up most of the language. Functions, types, variables, arithmetic expressions: all have paral-

lels in previous intellectual experiences beginners might have had, from elementary school maths to experimenting with simpler programming languages. But most novice C programmers hit a brick wall when they encounter pointers.

Pointers are second nature to experienced C programmers, most of whom consider their complete understanding necessary for the proper use of C. Because pointers are so fundamental, most experienced C programmers would not advise limits on their use for stylistic or learning purposes. Despite this, many C novices feel pointers are an unnecessary complication and avoid their use, resulting in the *FORTRAN in any language* symptom where valuable language features are neglected. The disease is ignorance of the language's features, not poor programming style. Once the features are fully understood, the correct styles are obvious. An auxiliary theme of this book, one that applies to any programming language, is that in programming, style is not something to pursue directly. Style is necessary only where understanding is missing[1].

Like C pointers, the macro is a feature of lisp that is often poorly understood, the wisdom on its proper use being very distributed and idealised. If when considering macros you find yourself relying on stylistic aphorisms like

> Macros change the syntax of lisp code.
>
> Macros work on the parse tree of your program.
>
> Only use macros when a function won't do.

you are probably missing the big picture when it comes to macro programming. That is what this book hopes to fix.

There are very few good references or tutorials on macro construction. Paul Graham's *On Lisp*[ON-LISP] is one of the exceptions. Every word of *On Lisp* is required reading for anyone interested in macros. *On Lisp* and Graham's other writings were the most

[1] A corollary to this is that sometimes the only way to effectively use something you don't understand is to copy styles observed elsewhere.

important inspirations for the creation of the book you are reading now. Thanks to Paul Graham and other lisp writers, the power that macros provide programmers is widely discussed, yet is unfortunately still widely misunderstood. Despite the wisdom regarding macro programming that can be gleaned from a simple perusal of *On Lisp*, few programmers make the connection between the macro and their real-life programming problems. While *On Lisp* will show you the different types of macros, this book will show you how to use them.

Macro writing is a reflective and iterative process. All complex macros come from simpler macros, often through a long series of improvement-test cycles. What's more, recognising where to apply macros is an acquired skill that comes directly from writing them. When you write a program, you, as a conscious human, are following a system and a process whether you are aware of it or not. Every programmer has a conceptual model of how programming tools work and the creation of code comes as a direct, logical result of this. Once an intelligent programmer begins to think of the act of programming as a logical procedure, the logical next step is for this process to benefit from automation itself. After all, programmers are trained to do exactly this: automate processes.

The crucial first step to understanding macros is to recognise that without careful planning and lots of effort, large portions of any programs will have redundant patterns and inflexible abstractions littered throughout. This can be seen in almost any large software project as duplicated code or as code that is needlessly complex because the right abstractions weren't available to its authors. The effective use of macros entails recognising these patterns and abstractions, and then creating *code to help you code.* It is not enough to understand how to write macros; a professional lisp programmer needs to know why to write macros.

C programmers who are new to lisp often make the mistake of assuming that the primary purpose of a macro is to improve the efficiency of code at run-time[2]. While macros are often very useful for this task, by far the most common use of a macro is to make

[2]C programmers make this mistake because they are used to a "macro system" that is good for little else.

the job of programming a desired application easier. Because large portions of the patterns in most programs are redundantly copied and the generality of their abstractions not fully exploited, properly designed macros can enable programming on literally new planes of expression. Where other languages are rigid and specific, lisp is fluid and generic.

This book is not an introduction to lisp. The topics and material are aimed at professional programmers of non-lisp languages who are curious as to what macros have to offer, and at intermediate lisp students who are ready to really learn what makes lisp special. Basic to intermediate knowledge of lisp programming is assumed, but a deep understanding of closures and macros is not.

This book is also not about theory. All examples involve working, usable code that can help improve your programming, today and now. This book is about using advanced programming techniques to help you program better. In contrast to many other programming books that deliberately use a simple programming style in an attempt to improve accessibility, this book takes the view that the best approach to teaching programming is full utilisation of the language. Although many of the provided code samples use esoteric features of COMMON LISP, such potentially unfamiliar features are described as they are used. For calibration, if you have read and understood[3] everything in *chapter 2, Closures* and *chapter 3, Macro Basics*, for the purposes of this book you can consider yourself past the intermediate stage of lisp understanding.

Part of lisp is discovering things yourself and this book will not deprive you of that. Be warned that this book moves more quickly than most, more quickly than you might be used to. To understand some of the code in this book you may need to consult additional COMMON LISP tutorials or references. After we cover the basics we will move directly into explaining some of the most advanced macro research to-date, much of which borders a large, unexplored gray-area of intellectual terrain. As does all advanced macro programming, this book focuses heavily on *combinations*

[3]Not necessarily agreed with, of course.

of macros. This topic has a frightening reputation and is well understood by few, if any, programmers. Combinations of macros represent the most vast and fertile area of research in programming languages today. Academia has squeezed out most of the interesting results from types, objects, and prolog-style logic, but macro programming remains a huge, gaping black hole. Nobody really knows what lies beyond. All we know is that, yes, it is complicated and frightening and currently appears boundless in potential. Unlike too many other programming ideas, the macro is neither an academic concept for churning out useless theoretical publications, nor an empty enterprise software buzzword. Macros are a hacker's best friend. Macros let you program smarter, not harder. Most programmers who come to understand macros decide they never again want to program without them.

While most lisp books are written to make lisp more popular, I am completely unconcerned with lisp's day-to-day public appeal. Lisp isn't going away. I would be perfectly happy if I could continue to use lisp as a *secret weapon* for the remainder of my programming career. If this book has only one purpose, it is to inspire the study and research of macros, just as I have been inspired by them in *On Lisp*. I hope readers of this book might also be so inspired that some day I might enjoy even better lisp macro tools and even more interesting lisp macro books.

Still in awe of lisp's power,
your humble author,
Doug Hoyte

1.2 U-Language

Since discussing macros involves discussing discussion itself, we need to be very clear about the conventions we are adopting for this book. What I am writing right now, as conveyed to you by what you are reading and interpreting, is itself a system of expression worth formalising and analysing.

Nobody has understood this better than Haskell Curry, the author of *Foundations Of Mathematical Logic*[FOUNDATIONS]. Curry, because he was not only trying to formalise ideas, but also the very

Listing 1.1: EXAMPLE-PROGRAM-LISTING

```
(defun example-program-listing ()
  '(this is
     (a (program
          (listing)))))
```

expression of ideas, found it necessary to abstract this concept of a communicative language between writer and reader. He called it the U-Language.

> Every investigation, including the present one, has to be communicated from one person to another by means of language. It is expedient to begin our study by calling attention to this obvious fact, by giving a name to the language being used, and by being explicit about a few of its features. We shall call the language being used the U-Language. [...] There would be no point in calling attention to it, if it were not for the fact that language is more intimately related to our job than of most others.

Throughout this book we will introduce key new concepts or points that otherwise deserve emphasis in *this special font.* When referencing special forms, functions, macros, and other identifiers found in a program, either presented or foreign, we will use `this special font` (notice that some words have multiple meanings, for example `lambda` the COMMON LISP macro versus lambda the concept; `let` the special form versus a list that is a let form).

In this book new pieces of code are introduced in the form of *program listings.* Code that is designed for re-use, or for an example of proper implementation, is presented as in the definition of our function `example-program-listing`. But sometimes we wish to demonstrate the use of a bit of code or just want to discuss properties of some expressions without departing the flow of the written text[4]. In those cases, the code, or example uses of the code, will appear like so:

[4] And this is a foot-note, a relevant but concise departure from the main

```
(this is
  (demonstration code))
```

Much writing that teaches programming makes heavy use of isolated, contrived examples to illustrate a point but forgets to tie it in with reality. This book's examples try to be as minimal and direct as possible in order to illustrate the big-picture programming ideas currently being explained. Some writing tries to hide being boring by using cute, quirky identifier names or skin-deep analogies in its examples. Our examples serve only to illustrate ideas. That said, above all this book tries not to take itself (or anything) too seriously. There is humour here, the difference is that you need to look for it.

Because of lisp's interactive nature, the results of evaluating a simple expression can often convey more than the equivalent quantity of U-Language. In such cases, this is how we will show the output from a COMMON LISP Read Evaluate Print Loop (called the *REPL*):

```
* (this is
    (the expression
      (to evaluate)))

THIS-IS-THE-RESULT
```

Notice how the text we enter is in lower-case but the text returned from lisp is in upper-case. This is a feature of COMMON LISP that allows us to easily scan a REPL print-out and know which expressions we entered versus which were printed out by lisp. More precisely, this feature lets us quickly scan any lisp form that contains symbols—in any file or on any screen—and instantly know whether it has yet been processed by the lisp reader. Also notice that the asterisk character (*) represents a prompt. This character is ideal because it can't be confused with a balanced character and because of its high pixel count that makes it stand out clearly when scanning a REPL session.

---

text.

Writing complicated lisp macros is an *iterative* process. Nobody sits down and hammers out a page-long macro in the cavalier style common to programs in other languages. This is partly because lisp code contains much more information per page than most other languages and also partly because lisp technique encourages programmers to grow their programs: refining them in a series of enhancements dictated by the needs of the application.

This book distinguishes types of lisp, like COMMON LISP and Scheme, from the more abstract notion of lisp the building material. Another important distinction is made between lisp programming languages and non-lisp programming languages. Sometimes we need to talk about non-lisp languages and, to make as few enemies as possible, would like to avoid picking on any language in particular. To do so, we resort to the following unusual definition:

> A language without lisp macros is a *Blub.*

The U-language word Blub comes from an essay by Paul Graham, *Beating the Averages*[BEATING-AVGS], where Blub is a hypothetical language used to highlight the fact that lisp is not like other languages: lisp is different. Blub is characterised by infix syntax, annoying type systems, and crippled object systems but its only unifying trait is its lack of lisp macros. Blub terminology is useful to us because sometimes the easiest way to understand an advanced macro technique is to consider why the technique is impossible in Blub. The purpose of Blub terminology is not to poke fun at non-lisp languages[5].

In order to illustrate the iterative process of macro creation, this book adopts the convention where the percent (%) character is appended to the names of functions and macros whose definitions are incomplete or are yet to be improved upon in some other way. Multiple revisions can result in multiple % characters on the end of a name before we settle on the final version with no % characters.

Macros are described in Curry's terminology as *meta-programming.* A meta-program is a program with the sole

[5]There will be a little bit of fun.

Listing 1.2: ITERATIVE-PROCESS-EXAMPLE

```
(defun example-function% () ; first try
  t)

(defun example-function%% () ; second try
  t)

(defun example-function () ; got it!
  t)
```

purpose of enabling a programmer to better write programs. Although meta-programming is adopted to various extents in all programming languages, no language adopts it as completely as lisp. In no other language is the programmer required to write code in such a way to convenience meta-programming techniques. This is why lisp programs look *weird* to non-lisp programmers: how lisp code is expressed is a direct consequence of its meta-programming needs. As this book attempts to describe, this design decision of lisp—writing meta-programs in lisp itself—is what gives lisp the stunning productivity advantages that it does. However, because we create meta-programs in lisp, we must keep in mind that meta programming is different from U-Language specification. We can discuss meta-languages from different perspectives, including other meta-languages, but there is only one U-Language. Curry makes this clear for his system as well:

> We can continue to form hierarchies of languages with any number of levels. However, no matter how many levels there are, the U-Language will be the highest level: if there are two levels, it will be the meta-language; if there are three levels, it will be the meta-meta-language; and so on. Thus the terms U-Language and meta-language must be kept distinct.

This is a book about lisp, of course, and lisp's logic system is very different than that described by Curry so we will adopt

few other conventions from his work. But Curry's contributions to logic and meta-programming continue to inspire us to this day. Not only because of his profound insights regarding symbolic quotation, but also his beautifully phrased and executed U-Language.

1.3 The Lisp Utility

On Lisp is one of those books that you either understand or you don't understand. You either adore it or you fear it. Starting with its very title, *On Lisp* is about creating programming abstractions which are layers *on top of lisp*. After we've created these abstractions we are free to create more programming abstractions which are successive layers on earlier abstractions.

In almost any language worth using, large portions of the language's functionality is implemented with the language itself; Blub languages usually have extensive standard libraries written in Blub. When even implementors don't want to program in the target language, you probably won't want to either.

But even after considering the standard libraries of other languages, lisp is different. In the sense that other languages are composed of primitives, lisp is composed of meta-primitives. Once macros are standardised, as in COMMON LISP, the rest of the language can be *boot-strapped* up from essentially nothing. While most languages just try to give a flexible enough set of these primitives, lisp gives a meta-programming system that allows any and all sorts of primitives. Another way to think about it is that lisp does away with the concept of primitives altogether. In lisp, the meta-programming system doesn't stop at any so-called primitives. It is possible, in fact desired, for these macro programming techniques used to build the language to continue on up into the user application. Even applications written by the highest-level of users are still macro layers on the lisp onion, growing through iterations.

In this light, there being primitives in a language at all is a problem. Any time there is a primitive, there is a barrier, a non-orthogonality, in the design of the system. Sometimes, of course, this is warranted. Most programmers have no problem

Listing 1.3: MKSTR-SYMB

```
(defun mkstr (&rest args)
  (with-output-to-string (s)
    (dolist (a args) (princ a s))))

(defun symb (&rest args)
  (values (intern (apply #'mkstr args))))
```

treating individual machine code instructions as primitives for their C or lisp compilers to handle. But lisp users demand control over nearly everything else. No other languages are, with respect to the control given to the programmer, as complete as lisp.

Heeding the advice of *On Lisp*, the book you are currently reading was itself designed as another layer on the onion. In the same sense that programs are layered on other programs, this book is layered on *On Lisp*. It is the central theme of Graham's book: well-designed *utilities* can, when combined, work together to give a greater than the sum of the parts productivity advantage. This section describes a collection of useful utilities from *On Lisp* and elsewhere.

`Symb`, layered upon `mkstr`, is a general way of creating symbols. Since symbols can be referenced by any arbitrary string, and creating symbols programmatically is so useful, `symb` is an essential utility for macro programming and is used heavily throughout this book.

`Group` is another utility that consistently pops up when writing macros. Part of this is because of the need to mirror operators like COMMON LISP's `setf` and `psetf` that already group arguments, and part of it is because grouping is often the best way to structure related data. Since we use this functionality so often, it makes sense to make the abstraction as general as possible. Graham's `group` will group by any provided grouping amount, specified by the parameter `n`. In cases like `setf`, where the arguments are grouped into pairs, `n` is 2.

`Flatten` is one of the most important utilities in *On Lisp*. Given an arbitrarily nested list structure, `flatten` will return a

Listing 1.4: GROUP

```
(defun group (source n)
  (if (zerop n) (error "zero length"))
  (labels ((rec (source acc)
             (let ((rest (nthcdr n source)))
               (if (consp rest)
                   (rec rest (cons
                              (subseq source 0 n)
                              acc))
                   (nreverse
                     (cons source acc))))))
    (if source (rec source nil) nil)))
```

Listing 1.5: FLATTEN

```
(defun flatten (x)
  (labels ((rec (x acc)
             (cond ((null x) acc)
                   ((atom x) (cons x acc))
                   (t (rec
                        (car x)
                        (rec (cdr x) acc))))))
    (rec x nil)))
```

Listing 1.6: FACT-AND-CHOOSE

```
(defun fact (x)
  (if (= x 0)
    1
    (* x (fact (- x 1)))))

(defun choose (n r)
  (/ (fact n)
     (fact (- n r))
     (fact r)))
```

new list containing all the atoms reachable through that list structure. If we think of the list structure as being a tree, `flatten` will return a list of all the leaves in the tree. If that tree represents lisp code, by checking for the presence of certain objects in an expression, `flatten` accomplishes a sort of *code-walking*, a recurring theme throughout this book.

`Fact` and `choose` are the obvious implementations of the factorial and binomial coefficient functions.

1.4 License

Because I believe the concepts behind the code presented in this book are as fundamental as physical observations or mathematical proofs, even if I wanted to I don't believe I could claim their ownership. For that reason you are basically free to do whatever you want with the code from this book. Here is the very liberal license distributed with the code:

```
;; This is the source code for the book
;; _Let_Over_Lambda_ by Doug Hoyte.
;; This code is (C) 2002-2008, Doug Hoyte.
;;
;; You are free to use, modify, and re-distribute
;; this code however you want, except that any
;; modifications must be clearly indicated before
;; re-distribution. There is no warranty,
```

```
;; expressed nor implied.
;;
;; Attribution of this code to me, Doug Hoyte, is
;; appreciated but not necessary. If you find the
;; code useful, or would like documentation,
;; please consider buying the book!
```

The text of this book is (C) 2008 Doug Hoyte. All rights reserved.

1.5 Thanks

Brian Hoyte, Nancy Holmes, Rosalie Holmes, Ian, Alex, all the rest of my family; syke, madness, fyodor, cyb0rg/asm, theclone, blackheart, d00tz, rt, magma, nummish, zhivago, defrost; Mike Conroy, Sylvia Russell, Alan Paeth, Rob McArthur, Sylvie Desjardins, John McCarthy, Paul Graham, Donald Knuth, Leo Brodie, Bruce Schneier, Richard Stallman, Edi Weitz, Peter Norvig, Peter Seibel, Christian Queinnec, Keith Bostic, John Gamble; the designers and creators of COMMON LISP, especially Guy Steele, Richard Gabriel, and Kent Pitman, the developers and maintainers of CMUCL/SBCL, CLISP, OpenBSD, GNU/Linux.

Special thanks to Ian Hoyte for the cover design and Leo Brodie for the back-cover cartoon.

This book is dedicated to everyone who loves programming.

Chapter 2

Closures

2.1 Closure-Oriented Programming

> One of the conclusions that we reached was that the "object" need not be a primitive notion in a programming language; one can build objects and their behaviour from little more than assignable value cells and good old lambda expressions.
>
> —Guy Steele on the design of Scheme

Sometimes it's called a *closure*, other times a saved lexical environment. Or, as some of us like to say, *let over lambda.* Whatever terminology you use, mastering this concept of a closure is the first step to becoming a professional lisp programmer. In fact, this skill is vital for the proper use of many modern programming languages, even ones that don't explicitly contain let or lambda, such as Perl or Javascript.

Closures are one of those few curious concepts that are paradoxically difficult because they are so simple. Once a programmer becomes used to a complex solution to a problem, simple solutions to the same problem feel incomplete and uncomfortable. But, as we will soon see, closures can be a simpler, more direct solution to the problem of how to organise data and code than objects. Even more important than their simplicity, closures represent a

better abstraction to use when constructing macros—the topic of this book.

The fact that we can build objects and classes with our closure primitives doesn't mean that object systems are useless to lisp programmers. Far from it. In fact, COMMON LISP includes one of the most powerful object systems ever devised: *CLOS*, the COMMON LISP Object System. Although I am very impressed with the flexibility and features of CLOS, I seldom find a need to use its more advanced features[1], thanks to assignable value cells and good old lambda expressions.

While much of this book assumes a reasonable level of lisp skill, this chapter attempts to teach the theory and use of closures from the very basics as well as to provide a common terminology for closures that will be used throughout the rest of this book. This chapter also examines the efficiency implications of closures and considers how well modern compilers optimise them.

2.2 Environments and Extent

What Steele means by assignable value cells is an environment for storing pointers to data where the environment is subject to something called *indefinite extent*. This is a fancy way of saying that we can continue to refer to such an environment at any time in the future. Once we allocate this environment, it and its references are there to stay as long as we need them. Consider this C function:

```
#include <stdlib.h>

int *environment_with_indefinite_extent(int input) {
  int *a = malloc(sizeof(int));
  *a = input;
  return a;
}
```

[1] CLOS is so central to COMMON LISP that it is literally impossible to program in COMMON LISP without it.

After we call this function and receive the pointer it returns, we can continue to refer to the allocated memory indefinitely. In C, new environments are created when invoking a function, but C programmers know to `malloc()` the required memory when returning it for use outside the function.

By contrast, the example below is flawed. C programmers consider `a` to be automatically collected when the function returns because the environment is allocated on the *stack*. In other words, according to lisp programmers, `a` is allocated with *temporary extent*.

```
int *environment_with_temporary_extent(int input) {
  int a = input;
  return &a;
}
```

The difference between C environments and lisp environments is that unless you explicitly tell lisp otherwise it always assumes you mean to use indefinite extent. In other words, lisp always assumes you mean to call `malloc()` as above. It can be argued that this is inherently less efficient than using temporary extent, but the benefits almost always exceed the marginal performance costs. What's more, lisp can often determine when data can safely be allocated on the stack and will do so automatically. You can even use *declarations* to tell lisp to do this explicitly. We will discuss declarations in more detail in *chapter 7, Macro Efficiency Topics.*

But because of lisp's dynamic nature, it doesn't have explicit pointer values or types like C. This can be confusing if you, as a C programmer, are used to casting pointers and values to indicate types. Lisp thinks about all this slightly differently. In lisp, a handy mantra is the following:

> Variables don't have types. Only values have types.

Still, we have to return something to hold pointers. In lisp there are many data structures that can store pointers. One of the most favoured by lisp programmers is a simple structure: the

cons cell. Each cons cell holds exactly two pointers, affectionately called car and cdr. When `environment-with-indefinite-extent` is invoked, a cons cell will be returned with the car pointing to whatever was passed as `input` and the cdr pointing to `nil`. And, most importantly, this cons cell (and with it the pointer to `input`) has indefinite extent so we can continue to refer to it as long as we need to:

```
(defun environment-with-indefinite-extent (input)
  (cons input nil))
```

The efficiency disadvantages of indefinite extent are approaching irrelevance as the state of the art in lisp compilation technology improves. Environments and extent are closely related to closures and more will be said about them throughout this chapter.

2.3 Lexical and Dynamic Scope

The technical term for where to consider a variable reference valid is *scope.* The most common type of scope in modern languages is called *lexical* scope. When a fragment of code is surrounded by the lexical binding of a variable, that variable is said to be in the lexical scope of the binding. The `let` form, which is one of the most common ways to create bindings, can introduce these lexically scoped variables:

```
* (let ((x 2))
    x)

2
```

The `x` inside the body of the `let` form was accessed through lexical scope. Similarly, arguments to functions defined by `lambda` or `defun` are also lexically bound variables inside the text of the function definition. Lexical variables are variables that can only be accessed by code appearing inside the context of, for instance,

the above `let` form. Because lexical scoping is such an intuitive way to limit the scope of access to a variable, it can appear to be the only way. Are there any other possibilities for scoping?

As useful as the combination of indefinite extent and lexical scoping turns out to be, it has until recently not been used to its fullest extent in mainstream programming languages. The first implementation was by Steve Russell for Lisp 1.5[HISTORY-OF-LISP] and was subsequently designed directly into languages like Algol-60, Scheme, and COMMON LISP. Despite this long and fruitful history, the numerous advantages of lexical scoping are only slowly being taken up by many Blubs.

Although the scoping methods provided by C-like languages are limited, C programmers need to program across different environments too. To do so, they often use an imprecisely defined scoping known as *pointer scope*. Pointer scope is famous for its difficulty to debug, numerous security risks, and, somewhat artificially, its efficiency. The idea behind pointer scoping is to define a domain specific language for controlling the registers and memory of a Von Neumman machine similar to most modern CPUs[PAIP-PIX], then to use this language to access and manipulate data-structures with fairly direct commands to the CPU running the program. Pointer scoping was necessary for performance reasons before decent lisp compilers were invented but is now regarded as a problem with, rather than a feature of, modern programming languages.

Even though lisp programmers seldom think in terms of pointers, the understanding of pointer scoping is very valuable in the construction of efficient lisp code. In *section 7.4, Pointer Scope* we will investigate implementing pointer scoping for the rare cases where we need to instruct the compiler on specific code creation. But for now we only need discuss its mechanics. In C, we sometimes would like to access a variable defined outside the function we are writing:

```
#include <stdio.h>

void pointer_scope_test() {
```

```
  int a;
  scanf("%d", &a);
}
```

In the above function we use the C `&` operator to give the actual address in memory of our local variable `a` to the `scanf` function so it knows where to write the data it scans. Lexical scoping in lisp forbids us from implementing this directly. In lisp, we would likely pass an anonymous function to a hypothetical lisp `scanf` function, allowing it to set our lexical variable `a` even though `scanf` is defined outside our lexical scope:

```
(let (a)
  (scanf "%d" (lambda (v) (setf a v))))
```

Lexical scope is the enabling feature for closures. In fact, closures are so related to this concept of lexical scope that they are often referred to more specifically as *lexical closures* to distinguish them from other types of closures. Unless otherwise noted, all closures in this book are lexical.

In addition to lexical scope, COMMON LISP provides *dynamic scope*. This is lisp *slang* for the combination of temporary extent and global scope. Dynamic scoping is a type of scoping that is unique to lisp in that it offers a very different behaviour but shares an identical syntax with lexical scope. In COMMON LISP we deliberately choose to call attention to variables accessed with dynamic scope by calling them *special variables*. These special variables can be defined with `defvar`. Some programmers follow a convention of prefixing and postfixing special variable names with asterisks, like `*temp-special*`. This is called the *earmuff* convention. For reasons explained in *section 3.7, Duality of Syntax*, this book does not use earmuffs so our special variable declarations look like this:

```
(defvar temp-special)
```

When defined like this, `temp-special` will be designated special[2] but will not be initialised with a value. In this state, a

[2]We can also indicate the specialness of variables by using declarations to make them locally special.

special variable is said to be *unbound.* Only special variables can be unbound—lexical variables are always bound and thus always have values. Another way of thinking of this is that by default all symbols represent lexically unbound variables. Just as with lexical variables, we can assign a value to special variables with `setq` or `setf`. Some lisps, like Scheme, do not have dynamic scope. Others, like EuLisp[SMALL-PIECES-P46], use different syntax for accessing lexical versus special variables. But in COMMON LISP the syntax is shared. Many lispers consider this a feature. Here we assign a value to our special variable `temp-special`:

```
(setq temp-special 1)
```

So far, this special variable doesn't seem that special. It seems to be just another variable, bound in some sort of global namespace. This is because we have only bound it once—its default special global binding. Special variables are most interesting when they are re-bound, or *shadowed,* by new environments. If we define a function that simply evaluates and returns `temp-special`:

```
(defun temp-special-returner ()
  temp-special)
```

This function can be used to examine the value that lisp evaluates `temp-special` to be at the moment in time when it was called:

```
* (temp-special-returner)

1
```

This is sometimes referred to as evaluating the form in a *null lexical environment.* The null lexical environment obviously doesn't contain any lexical bindings. Here the value of `temp-special` returned is that of its global special value, 1. But if we evaluate it in a non-null lexical environment—one that contains a binding for our special variable—the specialness of `temp-special` reveals itself[3]:

[3]Because we create a dynamic binding we are not actually creating a lexical environment. It just looks that way.

```
* (let ((temp-special 2))
    (temp-special-returner))

2
```

Notice that the value 2 was returned, meaning that the `temp-special` value was taken from our `let` environment, not its global special value. If this still does not seem interesting, see how this cannot be done in most other conventional programming languages as exemplified by this piece of Blub pseudo-code:

```
int global_var = 0;

function whatever() {
  int global_var = 1;
  do_stuff_that_uses_global_var();
}

function do_stuff_that_uses_global_var() {
  // global_var is 0
}
```

While the memory locations or register assignments for lexical bindings are known at compile-time[4], special variable bindings are determined at run-time—in a sense. Thanks to a clever trick, special variables aren't as inefficient as they seem. A special variable actually always does refer to the same location in memory. When you use `let` to bind a special variable, you are actually compiling in code that will store a copy of the variable, over-write the memory location with a new value, evaluate the forms in the let body, and, finally, restore the original value from the copy.

Special variables are perpetually associated with the symbol used to name them. The location in memory referred to by a special variable is called the `symbol-value` cell of a symbol. This is in direct contrast to lexical variables. Lexical variables are only indicated with symbols at compile-time. Because lexical variables

[4]Sometimes lexical scoping is called "static scoping" for this reason.

can only be accessed from inside the lexical scope of their bindings, the compiler has no reason to even remember the symbols that were used to reference lexical variables so it will remove them from compiled code. We will stretch the truth of this statement in *section 6.7, Pandoric Macros.*

Although COMMON LISP does offer the invaluable feature of dynamic scope, lexical variables are the most common. Dynamic scoping used to be a defining feature of lisp but has, since COMMON LISP, been almost completely replaced by lexical scope. Since lexical scoping enables things like lexical closures (which we examine shortly), as well as more effective compiler optimisations, the superseding of dynamic scope is mostly seen as a good thing. However, the designers of COMMON LISP have left us a very transparent window into the world of dynamic scoping, now acknowledged for what it really is: special.

2.4 Let It Be Lambda

`Let` is a lisp special form for creating an environment with names (bindings) initialised to the results of evaluating corresponding forms. These names are available to the code inside the `let` body while its forms are evaluated consecutively, returning the result of the final form. Although what `let` does is unambiguous, how it does it is deliberately left unspecified. What `let` does is separated from how it does it. Somehow, `let` needs to provide a data structure for storing pointers to values.

Cons cells are undeniably useful for holding pointers, as we saw above, but there are numerous structures that can be used. One of the best ways to store pointers in lisp is to let lisp take care of it for you with the `let` form. With `let` you only have to name (bind) these pointers and lisp will figure out how best to store them for you. Sometimes we can help the compiler make this more efficient by giving it extra bits of information in the form of declarations:

```
(defun register-allocated-fixnum ()
  (declare (optimize (speed 3) (safety 0)))
```

```
(let ((acc 0))
  (loop for i from 1 to 100 do
    (incf (the fixnum acc)
          (the fixnum i)))
  acc))
```

For example, in `register-allocated-fixnum` we provide some hints to the compiler that allow it to sum the integers from 1 to 100 very efficiently. When compiled, this function will allocate the data in registers, eliminating the need for pointers altogether. Even though it seems we've asked lisp to create an indefinite extent environment to hold `acc` and `i`, a lisp compiler will be able to optimise this function by storing the values solely in CPU registers. The result might be this machine code:

```
; 090CEB52:         31C9              XOR ECX, ECX
;       54:         B804000000        MOV EAX, 4
;       59:         EB05              JMP L1
;       5B: L0:     01C1              ADD ECX, EAX
;       5D:         83C004            ADD EAX, 4
;       60: L1:     3D90010000        CMP EAX, 400
;       65:         7EF4              JLE L0
```

Notice that 4 represents 1 and 400 represents 100 because fixnums are shifted by two bits in compiled code. This has to do with *tagging*, a way to pretend that something is a pointer but actually store data inside it. Our lisp compiler's tagging scheme has the nice benefit that no shifting needs to occur to index word aligned memory[DESIGN-OF-CMUCL]. We'll get to know our lisp compiler better in *chapter 7, Macro Efficiency Topics.*

But if lisp determines that you might want to refer to this environment later on it will have to use something less transient than a register. A common structure for storing pointers in environments is an array. If each environment has an array and all the variable references enclosed in that environment are just references into this array, we have an efficient environment with potentially indefinite extent.

As mentioned above, `let` will return the evaluation of the last form in its body. This is common for many lisp special forms and macros, so common that this pattern is often referred to as an *implicit progn* due to the `progn` special form designed to do nothing but this[5]. Sometimes the most valuable thing to have a let form return is an anonymous function which takes advantage of the lexical environment supplied by the let form. To create these functions in lisp we use *lambda*.

Lambda is a simple concept that can be intimidating because of its flexibility and importance. The lambda from lisp and scheme owes its roots to Alonzo Church's logic system but has evolved and adapted into its altogether own lisp specification. Lambda is a concise way to repeatably assign temporary names (bindings) to values for a specific lexical context and underlies lisp's concept of a function. A lisp function is very different from the mathematical function description that Church had in mind. This is because lambda has evolved as a powerful, practical tool at the hands of generations of lispers, stretching and extending it much further than early logicians could have foreseen.

Despite the reverence lisp programmers have for lambda, there is nothing inherently special about the notation. As we will see, lambda is just one of many possible ways to express this sort of variable naming. In particular, we will see that macros allow us to customise the renaming of variables in ways that are effectively impossible in other programming languages. But after exploring this, we will return to lambda and discover that it is very close to the optimal notation for expressing such naming. This is no accident. Church, as dated and irrelevant as he might seem to our modern programming environment, really was on to something. His mathematical notation, along with its numerous enhancements in the hands of generations of lisp professionals, has evolved into a flexible, general tool[6].

Lambda is so useful that, like many of lisp's features, most

[5] `Progn` is actually also useful for clustering forms to give them all top-level behaviour.

[6] The classic example of a macro is an implementation of `let` as a lambda form. I will not bore you with that in this book.

modern languages are beginning to import the idea from lisp into their own systems. Some language designers feel that lambda is too lengthy, instead using `fn` or some other abbreviation. On the other hand, some regard lambda as a concept so fundamental that obscuring it with a lesser name is next to heresy. In this book, although we will describe and explore many variations on lambda, we happily call it lambda, just as generations of lisp programmers before us.

But what is lisp's lambda? First off, as with all names in lisp, lambda is a *symbol.* We can quote it, compare it, and store it in lists. Lambda only has a special meaning when it appears as the first element of a list. When it appears there, the list is referred to as a *lambda form* or as a *function designator.* But this form is not a function. This form is a list data structure that can be converted into a function using the `function` special form:

```
* (function '(lambda (x) (+ 1 x)))

#<Interpreted Function>
```

COMMON LISP provides us a convenience shortcut for this with the #' (sharp-quote) read macro. Instead of writing `function` as above, for the same effect we can take advantage of this shortcut:

```
* #'(lambda (x) (+ 1 x))

#<Interpreted Function>
```

As a further convenience feature, lambda is also defined as a macro that expands into a call to the function special form above. The COMMON LISP ANSI standard requires[ANSI-CL-ISO-COMPATIBILITY] a `lambda` macro defined like so:

```
(defmacro lambda (&whole form &rest body)
  (declare (ignore body))
  `#',form)
```

Ignore the ignore declaration for now[7]. This macro is just a simple way to automatically apply the `function` special form to your function designators. This macro allows us to evaluate function designators to create functions because they are expanded into sharp-quoted forms:

```
* (lambda (x) (+ 1 x))

#<Interpreted Function>
```

There are few good reasons to prefix your lambda forms with #' thanks to the `lambda` macro. Because this book makes no effort to support pre-ANSI COMMON LISP environments, backwards compatibility reasons are easily rejected. But what about stylistic objections? Paul Graham, in *ANSI* COMMON LISP[GRAHAM-ANSI-CL], considers this macro, along with its brevity benefits, a "specious sort of elegance at best". Graham's objection seems to be that since you still need to sharp-quote functions referenced by symbols, the system seems asymmetric. However, I believe that not sharp-quoting lambda forms is actually a stylistic improvement because it highlights the asymmetry that exists in the second namespace specification. Using sharp-quote for symbols is for referring to the second namespace, whereas functions created by lambda forms are, of course, nameless.

Without even invoking the `lambda` macro, we can use lambda forms as the first argument in a function call. Just like when a symbol is found in this position and lisp assumes we are referencing the `symbol-function` cell of the symbol, if a lambda form is found, it is assumed to represent an anonymous function:

```
* ((lambda (x) (+ 1 x)) 2)

3
```

But note that just as you can't call a function to dynamically return the symbol to be used in a regular function call, you can't

[7] A U-Language declaration.

call a function to return a lambda form in the function position. For both of these tasks, use either `funcall` or `apply`.

A benefit of lambda expressions that is largely foreign to functions in C and other languages is that lisp compilers can often optimise them out of existence completely. For example, although `compiler-test` looks like it applies an increment function to the number 2 and returns the result, a decent compiler will be smart enough to know that this function always returns the value 3 and will simply return that number directly, invoking no functions in the process. This is called *lambda folding*:

```
(defun compiler-test ()
  (funcall
    (lambda (x) (+ 1 x))
    2))
```

An important efficiency observation is that a compiled lambda form is a constant form. This means that after your program is compiled, all references to that function are simply pointers to a chunk of machine code. This pointer can be returned from functions and embedded in new environments, all with no function creation overhead. The overhead was absorbed when the program was compiled. In other words, a function that returns another function will simply be a constant time pointer return function:

```
(defun lambda-returner ()
  (lambda (x) (+ 1 x)))
```

This is in direct contrast to the `let` form, which is designed to create a new environment at run-time and as such is usually not a constant operation because of the garbage collection overhead implied by lexical closures, which are of indefinite extent.

```
(defun let-over-lambda-returner ()
  (let ((y 1))
    (lambda (x)
      (incf y x))))
```

Every time `let-over-lambda-returner` is invoked, it must create a new environment, embed the constant pointer to the code represented by the lambda form into this new environment, then return the resulting *closure*. We can use `time` to see just how small this environment is:

```
* (progn
    (compile 'let-over-lambda-returner)
    (time (let-over-lambda-returner)))

; Evaluation took:
;   ...
;   24 bytes consed.
;
#<Closure Over Function>
```

If you try to call compile on a closure, you will get an error saying you can't compile functions defined in non-null lexical environments[CLTL2-P677]. You can't compile closures, only the functions that create closures. When you compile a function that creates closures, the closures it creates will also be compiled[ON-LISP-P25].

The use of a let enclosing a lambda above is so important that we will spend the remainder of this chapter discussing the pattern and variations on it.

2.5 Let Over Lambda

Let over lambda is a nickname given to a lexical closure. Let over lambda more closely mirrors the lisp code used to create closures than does most terminology. In a let over lambda scenario, the last form returned by a `let` statement is a `lambda` expression. It literally looks like `let` is sitting on top of `lambda`:

```
* (let ((x 0))
    (lambda () x))

#<Interpreted Function>
```

Recall that the `let` form returns the result of evaluating the last form inside its body, which is why evaluating this let over lambda form produced a function. However, there is something special about the last form in the `let`. It is a `lambda` form with `x` as a *free variable*. Lisp was smart enough to determine what `x` should refer to for this function: the `x` from the surrounding lexical environment created by the `let` form. And, because in lisp everything is of indefinite extent by default, the environment will be available for this function to use as long as it needs it.

So lexical scope is a tool for specifying exactly where references to a variable are valid, and exactly what the references refer to. A simple example of a closure is a *counter*, a closure that stores an integer in an environment and increments and returns this value upon every invocation. Here is how it is typically implemented, with a let over lambda:

```
(let ((counter 0))
  (lambda () (incf counter)))
```

This closure will return 1 the first time it is called, 2 the subsequent time, and so on. One way of thinking about closures is that they are functions with *state*. These functions are not mathematical functions, but rather procedures, each with a little memory of its own. Sometimes data structures that bundle together code and data are called *objects*. An object is a collection of procedures and some associated state. Since objects are so closely related to closures, they can often be thought of as one and the same. A closure is like an object that has exactly one method: `funcall`. An object is like a closure that you can `funcall` in multiple ways.

Although closures are always a single function and its enclosing environment, the multiple methods, inner classes, and static variables of object systems all have their closure counterparts. One possible way to simulate multiple methods is to simply return multiple `lambdas` from inside the same lexical scope:

```
(let ((counter 0))
  (values
    (lambda () (incf counter))
```

```
    (lambda () (decf counter))))
```

This *let over two lambdas* pattern will return two functions, both of which access the same enclosing counter variable. The first increments it and the second decrements it. There are many other ways to accomplish this. One of which, `dlambda`, is discussed in *section 5.7, Dlambda.* For reasons that will be explained as we go along, the code in this book will structure all data using closures instead of objects. Hint: It has to do with macros.

2.6 Lambda Over Let Over Lambda

In some object systems there is a sharp distinction between objects, collections of procedures with associated state, and classes, the data structures used to create objects. This distinction doesn't exist with closures. We saw examples of forms you can evaluate to create closures, most of them following the pattern let over lambda, but how can our program create these objects as needed?

The answer is profoundly simple. If we can evaluate them in the REPL, we can evaluate them inside a function too. What if we create a function whose sole purpose is to evaluate a let over lambda and return the result? Because we use `lambda` to represent functions, it would look something like this:

```
(lambda ()
  (let ((counter 0))
    (lambda () (incf counter))))
```

When the *lambda over let over lambda* is invoked, a new closure containing a counter binding will be created and returned. Remember that `lambda` expressions are constants: mere pointers to machine code. This expression is a simple bit of code that creates new environments to close over the inner `lambda` expression (which is itself a constant, compiled form), just as we were doing at the REPL.

With object systems, a piece of code that creates objects is called a class. But lambda over let over lambda is subtly different

than the classes of many languages. While most languages require classes to be named, this pattern avoids naming altogether. Lambda over let over lambda forms can be called *anonymous classes.*

Although anonymous classes are often useful, we usually do name classes. The easiest way to give them names is to recognise that such classes are regular functions. How do we normally name functions? With the **defun** form, of course. After naming, the above anonymous class becomes:

```
(defun counter-class ()
  (let ((counter 0))
    (lambda () (incf counter))))
```

Where did the first **lambda** go? **Defun** supplies an *implicit lambda* around the forms in its body. When you write regular functions with **defun** they are still lambda forms underneath but this fact is hidden beneath the surface of the **defun** syntax.

Unfortunately, most lisp programming books don't provide realistic examples of closure usage, leaving readers with the inaccurate impression that closures are only good for toy examples like counters. Nothing could be further from the truth. Closures are the building blocks of lisp. Environments, the functions defined inside those environments, and macros like **defun** that make using them convenient, are all that are needed for modelling any problem. This book aims to stop beginning lisp programmers used to object-based languages from acting upon their gut instinct of reaching for systems like CLOS. While CLOS does have certain things to offer the professional lisp programmer, do not use it when a lambda will suffice.

In order to motivate the use of closures, a realistic example is presented: **block-scanner**. The problem **block-scanner** solves is that for some forms of data transfer the data is delivered in groups (blocks) of uncertain sizes. These sizes are generally convenient for the underlying system but not for the application programmer, often being determined by things like operating system buffers, hard drive blocks, or network packets. Scanning a stream of data for a specific sequence requires more than just scanning each block

Listing 2.1: BLOCK-SCANNER

```
(defun block-scanner (trigger-string)
  (let* ((trig (coerce trigger-string 'list))
         (curr trig))
    (lambda (data-string)
      (let ((data (coerce data-string 'list)))
        (dolist (c data)
          (if curr
            (setq curr
                  (if (char= (car curr) c)
                    (cdr curr) ; next char
                    trig))))   ; start over
        (not curr)))))  ; return t if found
```

as it comes in with a regular, stateless procedure. We need to keep state between the scanning of each block because it is possible that the sequence we are scanning for will be split between two (or more) blocks.

The most straightforward, natural way to implement this stored state in modern languages is with a closure. An initial sketch of a closure-based block scanner is given as `block-scanner`. Like all lisp development, creating closures is an iterative process. We might start off with code given in `block-scanner` and decide to improve its efficiency by avoiding coercion of strings to lists, or possibly improve the information gathered by counting the number of occurrences of the sequence.

Although `block-scanner` is an initial implementation waiting to be improved, it is still a good demonstration of the use of lambda over let over lambda. Here is a demonstration of its use, pretending to be some sort of communications tap watching out for a specific black-listed word, *jihad*:

```
* (defvar scanner
    (block-scanner "jihad"))

SCANNER
* (funcall scanner "We will start ")
```

```
NIL
# (funcall scanner "the ji")

NIL
* (funcall scanner "had tomorrow.")

T
```

2.7 Let Over Lambda Over Let Over Lambda

Users of object systems store values they want shared between all objects of a certain class into so-called *class variables* or *static variables*[8]. In lisp, this concept of sharing state between closures is handled by environments in the same way that closures themselves store state. Since an environment is accessible indefinitely, as long as it is still possible to reference it, we are guaranteed that it will be available as long as is needed.

If we want to maintain a global direction for all counters, up to increment each closure's counter and down to decrement, then we might want to use a let over lambda over let over lambda pattern:

```
(let ((direction 'up))
  (defun toggle-counter-direction ()
    (setq direction
          (if (eq direction 'up)
            'down
            'up)))

  (defun counter-class ()
    (let ((counter 0))
      (lambda ()
        (if (eq direction 'up)
```

[8]The term static is one of the most overloaded programming language terms. Variables shared by all objects of a class are called static variables in languages like Java, which is distantly related to one of the meanings of static in C.

```
    (incf counter)
    (decf counter))))))
```

In the above example, we have extended `counter-class` from the previous section. Now calling closures created with `counter-class` will either increment its counter binding or decrement it, depending on the value of the direction binding which is shared between all counters. Notice that we also take advantage of another `lambda` inside the direction environment by creating a function called `toggle-counter-direction` which changes the current direction for all counters.

While this combination of `let` and `lambda` is so useful that other languages have adopted it in the form of class or static variables, there exist other combinations of `let` and `lambda` that allow you to structure code and state in ways that don't have direct analogs in object systems[9]. Object systems are a formalisation of a subset of let and lambda combinations, sometimes with gimmicks like *inheritance* bolted on[10]. Because of this, lisp programmers often don't think in terms of classes and objects. Let and lambda are fundamental; objects and classes are derivatives. As Steele says, the "object" need not be a primitive notion in programming languages. Once assignable value cells and good old lambda expressions are available, object systems are, at best, occasionally useful abstractions and, at worst, special-case and redundant.

[9]But these analogs can sometimes be built on top of object systems.

[10]Having macros is immeasurably more important than having inheritance.

Chapter 3

Macro Basics

3.1 Iterative Development

> Lisp has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.
>
> —Edsger Dijkstra

The construction of a macro is an iterative process: all complex macros come from simpler macros. After starting with an idea, a rough implementation can be created from which the ultimate macro will emerge, like a sculpture from a block of stone. If the rough implementation isn't flexible enough, or results in inefficient or dangerous expansions, the professional macro programmer will slightly modify the macro, adding features or removing bugs until it satisfies all requirements.

The necessity of this iterative process for macro construction is partly because this is the most efficient programming style in general and partly because programming macros is more complicated than other types of programming. Because macro programming requires the programmer to think about multiple levels of code executed at multiple points in time, the complexity issues scale more rapidly than other types of programming. An iterative process helps ensure that your conceptual model is more closely aligned to what is actually being created than if the entire macro was written without this constant feedback.

In this chapter we will write some basic macros by introducing two common macro concepts: *domain specific languages* and *control structures.* Once these general macro areas are described, we take a step back and discuss the process of writing macros itself. Techniques like variable capture and free variable injection are introduced, along with the definition of a new, slightly more convenient syntax for defining lisp macros that is used throughout the remainder of this book.

3.2 Domain Specific Languages

COMMON LISP, along with most other programming environments, provides a function `sleep` which will pause execution of the process for `n` seconds, where `n` is a non-negative, non-complex, numeric argument. For instance, we might want to sleep for 3 minutes (180 seconds), in which case we could evaluate this form:

```
(sleep 180)
```

Or if we would rather think about sleeping in terms of minutes, we could instead use

```
(sleep (* 3 60))
```

Because compilers know how to *fold constants*, these two invocations are just as efficient. To be even more explicit about what we're doing, we might define a function `sleep-minutes`:

```
(defun sleep-minutes (m)
  (sleep (* m 60)))
```

Defining new functions for every unit of time that we might want to use is clunky and inconvenient. What we would really like is some sort of abstraction that lets us specify the unit of time along with the value. What we really want is a *domain specific language.*

So far, the lisp solution is the same as that of any other language: create a function that accepts a value and a unit and returns the value multiplied by some constant related to the given

Listing 3.1: SLEEP-UNITS-1

```
(defun sleep-units% (value unit)
  (sleep
    (* value
       (case unit
         ((s) 1)
         ((m) 60)
         ((h) 3600)
         ((d) 86400)
         ((ms) 1/1000)
         ((us) 1/1000000)))))
```

unit. But a possible lispy improvement becomes apparent when we consider our options for representing this unit. In languages like C it is customary to use an underlying data type like int and assign arbitrary values corresponding to the different units:

```
#define UNIT_SECONDS 1
#define UNIT_MINUTES 2
#define UNIT_HOURS 3

int sleep_units(int value, int unit) {
  switch(value) {
    case UNIT_SECONDS: return value;
    case UNIT_MINUTES: return value*60;
    case UNIT_HOURS: return value*3600;
  }
}
```

But in lisp the most obvious way to signal the desired unit is to use a symbol. A symbol in lisp exists mostly to be something not `eq` to other symbols. `Eq` is the fastest lisp comparison operator and roughly corresponds to a pointer comparison. Since pointers can be compared very quickly, symbols provide a very fast and convenient way to let two or more different lisp expressions know you're referring to the same thing. In lisp we might define the `sleep-units%` function so we can specify the units in our forms:

```
(sleep-units% 2 'm)
(sleep-units% 500 'us)
```

Because comparing symbols requires only a pointer comparison, **sleep-units%** will be compiled into a very fast run-time dispatch:

```
...
524:        CMP     ESI, [#x586FC4D0]     ; 'S
52A:        JEQ     L11
530:        CMP     ESI, [#x586FC4D4]     ; 'M
536:        JEQ     L10
538:        CMP     ESI, [#x586FC4D8]     ; 'H
53E:        JEQ     L9
540:        CMP     ESI, [#x586FC4DC]     ; 'D
546:        JEQ     L8
...
```

Notice how the unit given to **sleep-units%** must be quoted. This is because when lisp evaluates a function it first evaluates all arguments and then binds the results to variables for use inside the function. Numbers and strings and some other primitives evaluate to themselves which is why we don't need to quote the numeric values given to **sleep-units%**. But notice that they are evaluated so you are permitted to quote them if you like:

```
(sleep-units% '.5 'h)
```

Symbols, however, don't typically evaluate to themselves[1]. When lisp evaluates a symbol it assumes you are referring to a variable and tries to look up the value associated with that variable given your lexical context (unless the variable is declared special, in which case the dynamic environment).

To avoid quoting the unit, we need a macro. Unlike a function, a macro does not evaluate its arguments. Taking advantage of this fact, we replace the **sleep-units%** function with the **sleep-units** macro. Now we don't need to quote the unit:

[1] As a rule, no rule without exception. Some symbols do evaluate to themselves, for example: **t**, **nil**, and keywords.

Listing 3.2: SLEEP-UNITS

```
(defmacro sleep-units (value unit)
  `(sleep
     (* ,value
        ,(case unit
           ((s) 1)
           ((m) 60)
           ((h) 3600)
           ((d) 86400)
           ((ms) 1/1000)
           ((us) 1/1000000)))))
```

```
(sleep-units .5 h)
```

Although the main purpose of this macro is to avoid having to quote the `unit` argument, this macro is even more efficient than the function because there is no run-time dispatch at all: the unit and therefore the multiplier are known at compile time. Of course whenever we discover this sort of too-good-to-be-true situation, it probably really is too good to be true. This gain in efficiency isn't free. By foregoing the run-time dispatch we have lost the ability to determine the time unit at run-time. This makes it impossible to execute the following code using our macro:

```
(sleep-units 1 (if super-slow-mode 'd 'h))
```

This will not work because `sleep-units` expects the second argument to be one of the symbols in our case statement but instead it is a list with the first element the symbol `if`.

Recall that most macros are written to create more convenient and useful programming abstractions, not to improve the efficiency of the underlying code. Is it possible to extract any idioms from this code to make it more useful for the rest of our program (and possibly other future programs)? Even now we can foresee wanting to do other things with time values than just calling `sleep` on them. The macro `unit-of-time` abstracts functionality from the `sleep-units` macro, returning a value instead of calling `sleep` on it. The `value` parameter can be determined

Listing 3.3: UNIT-OF-TIME

```
(defmacro unit-of-time (value unit)
  `(* ,value
      ,(case unit
         ((s) 1)
         ((m) 60)
         ((h) 3600)
         ((d) 86400)
         ((ms) 1/1000)
         ((us) 1/1000000))))
```

at run-time because it is evaluated then, but `unit` cannot because we require the information at compile-time, just like `sleep-units`. Here is an example:

```
* (unit-of-time 1 d)

86400
```

Simple macros like `unit-of-time` provide a better syntax for solving a specific domain of problems and can give substantial productivity and correctness advantages. We will continue the development of this unit language further in *section 5.2, Top-Down Programming.* Unlike most programming languages, lisp gives you the same tools available to the people who created your programming environment. Macros are good enough for implementing the COMMON LISP language and they are good enough for implementing your own domain specific languages.

3.3 Control Structures

Although this book is focused on COMMON LISP, it is written for and about the Scheme programming language as well. Scheme is a wonderful language that, although lacking many features lisp programmers take for granted, still offers a flexible enough core for

Listing 3.4: NLET

```
(defmacro nlet (n letargs &rest body)
  `(labels ((,n ,(mapcar #'car letargs)
                ,@body))
     (,n ,@(mapcar #'cadr letargs))))
```

the professional lisp programmer to extend as necessary[2]. Similarly, there are a few features Scheme programmers rely on heavily that COMMON LISP doesn't specifically address. But comparisons between the features offered by each language are, with a few exceptions, meaningless. The gaps between the two languages can be, and frequently are, bridged. The bridges we use to cross between the two languages are, of course, macros.

Scheme's `let` form is in one respect more powerful than its COMMON LISP counterpart. Scheme's `let` form supports something called a *named let.* In Scheme, you can optionally insert a symbol before the bind list of a `let` form and Scheme will bind a function named by the provided symbol around the let body[3]. This function accepts new arguments for the values provided in the `let` bindings, providing a very convenient way to express loops.

Luckily we can build a bridge between Scheme and COMMON LISP with the `nlet` macro. `Nlet` lets us code in a Scheme style by emulating Scheme's named lets. In `nlet-fact`, `nlet` is used to define the factorial function by using a named let:

```
(defun nlet-fact (n)
  (nlet fact ((n n))
    (if (zerop n)
      1
      (* n (fact (- n 1))))))
```

Because `nlet` is one of our first macros, let's slow down and

[2]Scheme and COMMON LISP differ mostly in the communities they cater to. Scheme programmers like to talk about how great it is to have a short specification; COMMON LISP programmers like to write programs.

[3]Scheme only has one namespace so this function is bound there.

analyse it in depth. Sometimes to understand a macro it helps to *macroexpand* an example use of this macro[4]. To do that, provide a list representing this macro invocation to the `macroexpand` function. Notice that `macroexpand` will only expand macros that have their symbols in the first element of the list and will not expand nested macro invocations for you[5]. In the following, we've copied an invocation of `nlet` directly from `nlet-fact`, quoted it, and passed it to `macroexpand`:

```
* (macroexpand
    '(nlet fact ((n n))
       (if (zerop n)
         1
         (* n (fact (- n 1))))))

(LABELS ((FACT (N)
           (IF (ZEROP N)
             1
             (* N (FACT (- N 1))))))
  (FACT N))
T
```

The expansion uses the `labels` special form to bind a function around the provided body. The function is named according to the symbol used in the named let form. It takes as arguments the values bound with `nlet`, here only `n`. Since this function can be recursive, `nlet` implements a useful iteration construct.

Although simple macros might just be filling out backquote templates, most complicated macros at minimum make use of lisp's extensive list processing functions. `Mapcar`, applying a function to every element in a list and returning a list of the resulting values, turns up especially often in macros. Tellingly, `mapcar`

[4]The terminology of expansion is actually rather unfortunate. Nothing says that macroexpanding something will result in larger, expanded code. Sometimes forms even expand into nothing (ie `nil`).

[5]But `macroexpand` will continue to expand the macro until the first element no longer represents a macro. `Macroexpand-1` is useful for observing the first step in this process.

turns up often in regular lisp code as well. Lisp has been tuned to be as useful as possible for processing lists. In all kinds of lisp programming, including macro construction, we splice, merge, reduce, map, and filter lists. The only difference is that when programming macros, the output subsequently gets passed to a compiler or interpreter. Programming macros in lisp is actually the same process as programming regular lisp.

But what does it mean to say that `nlet` is a new control structure? A control structure is just a fancy way of describing some construct that doesn't follow the behaviour of a function. A function will evaluate each argument from left to right, bind the results in an environment, and execute machine code specified by some `lambda` form. Since `nlet` doesn't evaluate its arguments directly, instead splicing them into some chunk of lisp code, we have changed the flow of evaluation for `nlet` forms and thus have created a new control structure.

By this broad definition, virtually all macros—at least all interesting macros—define new control structures. When people say "only use macros when functions won't do", they mean that for any definition where you don't want to evaluate certain arguments, or you want to evaluate them out of order, or more than once, you will need to use a macro. Functions, no matter how cleverly written, simply will not work.

The `nlet` macro demonstrates one way that COMMON LISP was designed for macro writers. In binding forms such as `let`, it is a common convention for a variable to be bound to nil if no value is specified along with the variable name. In other words, `(let ((a)) a)` will be nil[6]. In Scheme, a language slightly less macro-writer friendly, this case must be checked for as a special case when iterating through such bindings because `(car nil)` and `(cdr nil)` raise type errors. In COMMON LISP, `(car nil)`, `(cdr nil)`, and therefore `(car (cdr nil))` and `(cadr nil)` are defined to return `nil`, allowing the second `mapcar` in `nlet` to work even if the empty let variable convention is used. This COMMON LISP feature is from Interlisp[INTERLISP].

[6]COMMON LISP even allows us to write `(let (a) a)` for the same effect.

Our `nlet` macro is different from Scheme's named lets in one subtle way. In this case, the interface to the macro is acceptable but the expansion may not be. As is common when programming across multiple levels, our mental model of the code can easily be slightly different from reality. In Scheme, a tail call of a named let is guaranteed to take up no additional stack space since Scheme is required, by the standard, to make this specific optimisation. This is not the case in COMMON LISP, however, so it is possible for stack overflows to occur in our COMMON LISP version of `nlet` that would not happen with named lets in Scheme. In *section 5.4, Code-Walking with Macrolet* we will see how to write a version of `nlet` with an identical interface but a potentially more efficient expansion[7].

3.4 Free Variables

A *free variable* is any variable or function referenced in an expression that doesn't have a global special binding or an enclosing lexical binding. In the following expression, `x` is free:

```
(+ 1 x)
```

But in the following, we create a binding around the form which *captures* the variable `x`, depriving it of its freedom:

```
(let ((x 1))
  (+ 1 x))
```

The terminology of freedom and capture may seem strange at first. After all, freedom implies consciousness and an ability to make decisions—something a simple expression is obviously lacking. But freedom doesn't refer to what the expression can do, rather what we, as programmers, can do with the expression. For example, we can take the expression `(+ 1 x)` and embed it anywhere we want, allowing our expression to access a binding

[7] In practice, this version of `nlet` is usually sufficient since a COMMON LISP compiler will almost certainly optimise tail calls in compiled code.

named `x` in the surrounding code. We then say that the code has *captured* our free variable. After the free variables in an expression are captured, as in the above `let` form, other surrounding code has no option of capturing our variable `x`. Our formerly free variable has already been captured. It is now completely unambiguous which `x` it refers to. Because of this, there isn't really any need for lisp to keep the reference to the symbol `x` in the code at all. As was described in detail in *section 2.3, Lexical and Dynamic Scope*, lisp compilers will forget the symbols that were used to represent lexical variables.

Although any language with expressions can have expressions with free variables, lisp's macro capabilities mean that free variables are much more useful in lisp than in other languages. In most languages we are forced to obey *referential transparency*. If there is no global or object variable `x` defined in a Blub program, the following code is unconditionally incorrect:

```
some_function_or_method() {
  anything(1 + x);
}
```

There is no way that `some_function_or_method` can create an *implicit binding* for `x`. In Blub, any use of a variable must have a textually apparent definition[8]. Languages with primitive macro systems (like C) can accomplish some of this in a very limited sense. But just as general purpose macros are impractical or impossible to write in C, so are the special cases involving free variables.

In lisp we can push around expressions with free variables as we please and either splice them into new expressions for them to be captured by surrounding code, or define global special variables to capture them. We can also write macros to modify which variables are free in an expression, either by re-writing the expression so it has fewer free variables (say by wrapping it in a let form, as above) or by modifying the expression in a way that adds

[8]Or, sometimes, in object oriented Blub, a class or object definition.

new free variables. Such addition of free variables is the opposite of capturing variables and is called *free variable injection.*

The simplest possible free variable injection is a macro that expands into a symbol reference:

```
(defmacro x-injector ()
  'x)
```

Because a macro is just a function, it executes its body as a regular lisp form. The above injector macro evaluates the quoted symbol and, of course, returns a symbol—a free variable—to be spliced into any expression that uses the `x-injector` macro. Discussing such free variable injection in *On Lisp*, Paul Graham writes

> This kind of lexical intercourse is usually viewed more as a source of contagion than a source of pleasure. Usually it would be bad style to write such a macro. Of all the macros in this book, only [two isolated cases] use the calling environment in this way.

By contrast, this book gets much pleasure from this sort of lexical intercourse. Free variable injection—writing a macro with full knowledge of the lexical environment it will be expanded in—is just another approach to lisp macro programming, one that is especially useful when there are a few slightly different lexical contexts that you would like to write mostly identical code inside. Although often the main advantage of a function call is that you throw out your lexical environment, sometimes, to lisp programmers, this is just a guide-line that can be ignored through the use of macros. In fact, once accustomed to it, some lisp programmers try to always write macros, extending the lexical context as far as possible, using a function only when they need to evaluate arguments or just chicken out and want a new lexical context. In *section 3.6, Once Only* we will see a way to avoid throwing out your lexical environment when you need arguments evaluated. Keeping the lexical environment around as much as possible allows for very interesting macro *combinations*, where a macro adds lexical

context around a use of one or more other macros. Expanding into code that uses the very macro being defined is a special case of macro combination and is treated in *section 5.5, Recursive Expansions.*

The shortest distance between two points is a straight line. Free variables and, more generally, extended lexical contexts are often the easiest way to programmatically construct a program. Using macros in this way might seem like a hack, and might feel objectionable on stylistic grounds, but it works conveniently and reliably. Especially after we consider `macrolet` in *section 5.4, Code-Walking with Macrolet*, this style of programming—combining macros—will begin to feel more comfortable. Just remember that macro programming is not about style; it is about power. Macros allow us to do many things that are impossible in other languages. Free variable injection is one of them.

3.5 Unwanted Capture

There are two perspectives on variable capture. Variable capture is the source of some very unpredictable bugs but when used properly can also be a highly desirable macro feature. Let's start our consideration of variable capture with a simple macro defined by Graham in *On Lisp*: `nif`. `Nif` is a *numeric if* which has four required clauses, compared to the regular boolean `if` that has two required clauses and an optional third clause. `Nif`, or rather the code that `nif` expands into, evaluates the first clause and assumes the result to be a non-complex number. It then evaluates one of the three respective clauses, depending on whether the result is positive (`plusp`), zero (`zerop`), or negative (otherwise). We can use it to test the variable `x` like so:

```
(nif x "positive" "zero" "negative")
```

`Nif` is the ideal function for our discussion of variable capture and we will use it to illustrate a few key points and also as a test case for a new notation for macro construction. Before we present the version of nif defined by Graham, let's define a nearly correct, but slightly buggy version:

```
(defmacro nif-buggy (expr pos zero neg)
  `(let ((obscure-name ,expr))
     (cond ((plusp obscure-name) ,pos)
           ((zerop obscure-name) ,zero)
           (t ,neg))))
```

`Nif-buggy` expands into a bit of code that uses `let` to bind the result of evaluating the user's supplied `expr` form. We need to do this because it is possible that evaluating `expr` will incur *side-effects* and we need to use its value for two separate things: passing it to `plusp` and passing it to `zerop`. But what do we call this temporary binding? To introduce a subtle bug we chose an arbitrary symbol, `obscure-name`. Unless someone looks at the macro expansion, nobody will ever see this name anyways, so it's no big deal, right?

`Nif-buggy` will appear to work like `nif` in almost all cases. As long as the symbol `obscure-name` is never used in the forms supplied to `nif-buggy`[9] then there is no possibility for unwanted variable capture. But what happens if `obscure-name` does appear in forms passed? In many cases, there is still no bug:

```
(nif-buggy
  x
  (let ((obscure-name 'pos))
    obscure-name)
  'zero
  'neg)
```

Even if `x` turns out to be positive, and even though we have injected the forbidden symbol into `nif-buggy`'s macroexpansion, this code still works as intended. When a new binding is created, and the references inside that binding always refer to the created binding, no unwanted variable capture occurs. The problem only appears when our usage of `obscure-name` *crosses over* its use in the expansion. Here is an example of unwanted variable capture:

[9] Or in the macro expansions of forms passed to it. See sub-lexical scope.

```
(let ((obscure-name 'pos))
  (nif-buggy
    x
    obscure-name
    'zero
    'neg))
```

In this case, `obscure-name` will be bound to the result of the evaluation of `x`, so the symbol `pos` will not be returned as was intended[10]. This is because our use of a symbol crossed over an invisible use of a binding. Sometimes code with invisible bindings like this is said to not be *referentially transparent.*

But isn't this just an academic issue? Surely we can think of rare enough names so that the problem never shows up. Yes, in many cases, packages and smart variable naming can solve the problem of variable capture. However, most serious variable capture bugs don't arise in code directly created by a programmer. Most variable capture problems only surface when other macros use your macro (combine with your macro) in ways you didn't anticipate. Paul Graham's has a direct answer for why to protect against unwanted variable capture:

> Why write programs with small bugs when you could write programs with no bugs?

I think we can distill the issue even further: no matter how subtle, why do something incorrectly when you can do it correctly?

Luckily, it turns out that variable capture, to the extent that it is a problem, is a solved problem with an easy solution. That last sentence is a controversial statement to many people, especially those who have decided they don't like the obvious solution and have dedicated large portions of time looking for a better one. As a professional macro programmer you will come into

[10] In truth, of course, this buggy behaviour was exactly what was intended. Very rarely are variable capture problems this direct and designed. More often they are surprising and subtle.

contact with many of these variable capture solutions. The current popular solution is to use so-called *hygienic macros*[11]. These solutions try to limit or eliminate the impact of unwanted variable capture but unfortunately do so at the expense of wanted, desirable variable capture. Almost all approaches taken to reducing the impact of variable capture serve only to reduce what you can do with `defmacro`. Hygienic macros are, in the best of situations, a beginner's safety guard-rail; in the worst of situations they form an electric fence, trapping their victims in a sanitised, capture-safe prison. Furthermore, recent research has shown that hygienic macro systems like those specified by various Scheme revisions can still be vulnerable to many interesting capture problems[SYNTAX-RULES-INSANE][SYNTAX-RULES-UNHYGIENIC].

The real solution to variable capture is known as the *generated symbol*, or *gensym* for short. A gensym is a way of having lisp pick the name of a variable for us. But instead of picking lame names like `obscure-name` as we did previously, lisp picks good names. Really good names. These names are so good and unique that there is no way anyone (even `gensym` itself) will ever pick the same names again. How is this possible? In COMMON LISP, symbols (names) are associated with *packages*. A package is a collection of symbols from which you can get pointers to by providing strings, their `symbol-name` strings. The most important property of these pointers (usually just called symbols) is that they will be `eq` to all other pointers (symbols) that have been looked up in that package with that same `symbol-name`. A gensym is a symbol that doesn't exist in any package, so there is no possible `symbol-name` that will return a symbol `eq` to it. Gensyms are for when you want to indicate to lisp that some symbol should be `eq` to some other symbol in an expression without having to name anything at all. Because you aren't naming anything, name collisions just can't happen.

So by following these three simple, very important rules, avoiding unwanted variable capture in COMMON LISP is easy:

Whenever you wrap a lexical or dynamic binding

[11] Another popular term used to be "macros by example".

> around code provided to your macro, name this binding with a gensym unless you want to capture it from the code you are wrapping.
>
> Whenever you wrap a function binding or a `macrolet` or `symbol-macrolet` macro around code provided to your macro, name this function or macro with a gensym unless you want to capture it from the code you are wrapping. Verify that this binding doesn't conflict with any of the special forms, macros, or functions defined by the standard.
>
> Never assign or re-bind a special form, macro, or function specified by COMMON LISP.

Some lisps other than COMMON LISP, like Scheme, have the unfortunate property of combining the variable namespace with the function/macro namespace. Sometimes these lisps are termed *lisp-1* lisps, while COMMON LISP, with its separate namespaces, is termed a *lisp-2* lisp. With a hypothetical *lisp-1* COMMON LISP we would also be obliged to follow these two additional rules when constructing macros:

> Verify that intentionally introduced lexical or dynamic bindings do not collide with intentionally introduced function or macro bindings, or any of the special forms, macros, or functions defined by the standard.
>
> Verify that intentionally introduced function or macro bindings do not collide with intentionally introduced lexical or dynamic bindings.

COMMON LISP's wise design decision to separate the variable namespace from the function namespace eliminates an entire dimension of unwanted variable capture problems. Of course lisp-1 lisps do not suffer any theoretical barrier to macro creation: if we follow the previous two rules, we can avoid variable capture in the same way as we do in COMMON LISP. However, when programming sophisticated macros it can be hard enough to keep track

of symbols in a single, isolated namespace. Having any cross-pollination of names to consider just makes macro writing more difficult than it needs to be.

More so than any other property except possibly its incomplete standard[12], it is this defect of a single namespace that makes Scheme, an otherwise excellent language, unfit for serious macro construction[13]. Richard Gabriel and Kent Pitman summarise the issue with the following memorable quote[LISP2-4LIFE]:

> There are two ways to look at the arguments regarding macros and namespaces. The first is that a single namespace is of fundamental importance, and therefore macros are problematic. The second is that macros are fundamental, and therefore a single namespace is problematic.

Because there is little of less importance than the quantity of namespaces, and little of more importance than the enabling of macros, it can only be concluded that Scheme made the *wrong* decision and COMMON LISP made the *right* decision.

Still, calling **gensym** every single time we want a nameless symbol is clunky and inconvenient. It is no wonder that the Scheme designers have experimented with so-called *hygienic* macro systems to avoid having to type **gensym** all over the place. The wrong turn that Scheme took was to promote a domain specific language for the purpose of macro construction. While Scheme's mini-language is undeniably powerful, it misses the entire point of macros: macros are great because they are written in lisp, not some dumbed down pre-processor language.

This book presents a new syntax for gensyms that should be more palatable to the brevity-conscious yet remains a thin film over traditional lisp expressions. Our new notation for gensyms, which we will use as the foundation for most of the macros in this book, is most clearly described by peeling off the layers of a

[12] Especially as it relates to macros and exceptions.

[13] Though as we will see throughout this book, there are many reasons to prefer COMMON LISP over Scheme.

simple macro which uses the features our notation offers. Let's continue with the `nif` example from the previous section. Here is how Graham defines a capture-safe `nif`:

```
(defmacro nif (expr pos zero neg)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ((plusp ,g) ,pos)
             ((zerop ,g) ,zero)
             (t ,neg)))))
```

This is how to use `gensym` correctly. As we saw in the previous section, a macro that can expand user input into something that could interfere with one of its variables must take care against variable capture. Graham presents a macro abbreviation `with-gensyms` that is somewhat more concise for situations where a number of gensyms need to be created:

```
(with-gensyms (a b c)
  ...)
```

Expands into

```
(let ((a (gensym))
      (b (gensym))
      (c (gensym)))
  ...)
```

Because needing gensyms in a `defmacro` form is so common, we decide to pursue the abbreviation further. In particular, notice that we have to type the temporary name for each of the gensyms (like `a`, `b`, and `c`) at least twice: once when we declare it a gensym and again when we use it. Can we eliminate this redundancy?

First, consider how the `nif` macro uses gensyms. When the `nif` macro is expanded, it calls `gensym` which returns a generated symbol. Because this symbol is guaranteed to be unique, we can safely splice it into a macro expansion knowing that it will never capture any unintended references. But we still need to name this

gensym in the definition of the macro so we are able to splice it into the expansion in the right places. Graham, for the scope of the `nif` macro definition, names this gensym `g`. Notice that this name never actually appears in the macro expansion of `nif`:

```
* (macroexpand '(nif x 'pos 'zero 'neg))

(LET ((#:G1605 X))
  (COND ((PLUSP #:G1605) 'POS)
        ((ZEROP #:G1605) 'ZERO)
        (T 'NEG)))
T
```

The name `g` disappears in the macro expansion. Because `g` was only bound in our expander environment, the name given to such a variable is irrelevant with respect to capture in the expansions. All occurrences of `g` have, in the expansion, been replaced by a symbol with a print name of `G1605`. It is prefixed by `#:` because the symbol is not *interned* in any package—it is a gensym. When printing out forms, it is necessary to prefix gensyms in such a way because we want lisp to break if we ever use (evaluate) this form after reading it back in again. We want lisp to break because we can't know by looking at the print names of two gensyms if they should be `eq` or not—that is their purpose. Lisp breaks in an interesting way: because each time a `#:` symbol is read in a new symbol is created, and because `(eq '#:a '#:a)` is never true, the inner `#:G1605` symbols in the above expansion do not refer to the binding created by the `let` form so lisp considers the expression to have a free variable, indicating to us that a form with gensyms was read back in again.

Despite the default printing behaviour for such uninterned symbols, it is still possible to save and reload macro expansions. For a more accurate printed representation of a form with gensyms, we can turn on `*print-circle*` mode when we print the results[14]:

[14]We return `t` so that we don't see the form returned by `print`. Returning `(values)` is also common.

Listing 3.5: G-BANG-SYMBOL-PREDICATE

```
(defun g!-symbol-p (s)
  (and (symbolp s)
       (> (length (symbol-name s)) 2)
       (string= (symbol-name s)
                "G!"
                :start1 0
                :end1 2)))
```

```
* (let ((*print-circle* t))
    (print
      (macroexpand '(nif x 'pos 'zero 'neg)))
    t)

(LET ((#1=#:G1606 X))
  (COND ((PLUSP #1#) 'POS)
        ((ZEROP #1#) 'ZERO)
        (T 'NEG)))
T
```

In the above form, the lisp printer uses the #= and ## *read macros.* These read macros allow us to create *self-referential* forms which we will discuss in more depth in *section 4.5, Cyclic Expressions.* If we read in the above form, the symbols used inside will actually be the same as the symbol used in the `let` binding and the expansion will still work. It seems as though the above definition has avoided the dual-naming redundancy. Is there a way we can pull this back up into a macro writing macro template?

Remember that we can name our gensyms anything in the macro definition, even, as Graham does, simple names like `g`, and they will disappear in the macro expansion. Because of this freedom in naming, let's standardise on a naming convention for gensyms. As a compromise between brevity and uniqueness, any symbol that starts with the two characters `G!`, and is followed by at least one other character is considered to be a special gensym referencing symbol called a *G-bang symbol.* We define a predicate, `g!-symbol-p`, which is a predicate for determining whether a given

Listing 3.6: DEFMACRO-WITH-G-BANG

```
(defmacro defmacro/g! (name args &rest body)
  (let ((syms (remove-duplicates
               (remove-if-not #'g!-symbol-p
                              (flatten body)))))
    `(defmacro ,name ,args
       (let ,(mapcar
              (lambda (s)
                `(,s (gensym ,(subseq
                               (symbol-name s)
                               2))))
              syms)
         ,@body))))
```

atom is a G-bang symbol.

Now that we have G-bang symbols standardised, we can create a macro that writes macro definitions for us and exploits a macro writing shortcut known as *automatic gensyms*. The macro `defmacro/g!` defines a domain specific language for the domain of macro writing, but manages to retain all of lisp's power. `Defmacro/g!` is simple, but how to use it, and how it works, may be non-obvious. Because of this, and because this is one of the first real macros we've presented in this book, we take the analysis of `defmacro/g!` slowly.

When dissecting any macro, the first step is to *stop*. Don't think of a macro as a syntax transformation or any other such nonsense abstraction. Think of a macro as a function. A macro is a function underneath, and works in the exact same way. The function is given the unevaluated expressions provided to it as arguments and is expected to return code for lisp to insert into other expressions.

So, thinking about `defmacro/g!` as a function, consider its execution. Because we are programming a regular lisp function, we have access to all of lisp's features, even utilities we've since added to the language. In `defmacro/g!`, we use Graham's `flatten` utility, lisp's `remove-if-not` and `remove-duplicates` functions, and our G-bang symbol predicate `g!-symbol-p` to create a new list

consisting of all the G-bang symbols found inside the body form that was passed to our macro. Next, we use a backquote template to return a list representing the code we would like the macro to expand into. In our case, because we're writing an improvement to `defmacro`, we would like our code to expand to a defmacro form itself. But we are adding new convenience features to the defmacro language and want to create a slightly more sophisticated expansion. In order to give each G-bang symbol found in the macro's body a fresh gensym, we use `mapcar` to map a function over the list of collected G-bang symbols, creating a new list that can be spliced into the `let` form, establishing bindings for each gensym[15].

Notice how the lambda that we map contains an expression created with the backquote operator, resulting in what appears to be—but is not—a *nested backquote* situation. Because the `mapcar` that applies this function is *unquoted*, the unquoted expressions in the nested backquote are still evaluated in our original context. Nested backquotes are notoriously difficult to understand and we will return to this concept when we look at backquote in more depth in *chapter 4, Read Macros.*

So what, exactly, does `defmacro/g!` let us do? It lets us exploit this technique of automatic gensyms, a way of checking for the presence of particular symbols in the lexical scope of code provided to the macro[16]. If we don't use any G-bang symbols, we can use `defmacro/g!` exactly like `defmacro`. But any G-bang symbols that occur in the body of the macro expansion are interpreted to mean:

> I want a gensym to be bound around this expression,
> and I've already given the symbol. Make it happen.

We can use this to save having to explicitly create a gensym in this re-definition of `nif`:

[15]The **gensym** function can optionally be passed a single string argument. This changes the gensym's print name which is helpful when reading expansions. `Defmacro/g!` uses the print name of the characters in the G-bang symbol for this purpose.

[16]This is, for now, a simplification. See the section on sub-lexical scope.

```
(defmacro/g! nif (expr pos zero neg)
  `(let ((,g!result ,expr))
     (cond ((plusp ,g!result) ,pos)
           ((zerop ,g!result) ,zero)
           (t ,neg))))
```

When we want to use a gensym we just use it. We need to be careful, of course, that all references to G-bang symbols are only evaluated by the macro expansion because that is the only place where the gensym will be bound[17]. Unquoting the G-bang symbols that occur inside a backquote, like above, is the most obvious way to do this, and we can see the direct parallel to the unquoting of the symbol g in Graham's original definition of nif.

So we have defined a macro nif that appears to function the same as Graham's, but this improvement almost seems too good to be true. Does it really work? Let's look at the macro expansion[18] before we decide:

```
* (macroexpand-1
    '(defmacro/g! nif (expr pos zero neg)
       `(let ((,g!result ,expr))
          (cond ((plusp ,g!result) ,pos)
                ((zerop ,g!result) ,zero)
                (t ,neg)))))

(DEFMACRO NIF (EXPR POS ZERO NEG)
  (LET ((G!RESULT (GENSYM "RESULT")))
    `(LET ((,G!RESULT ,EXPR))
       (COND ((PLUSP ,G!RESULT) ,POS)
             ((ZEROP ,G!RESULT) ,ZERO)
             (T ,NEG)))))
T
```

It seems that defmacro/g! wrote essentially the same code that Graham did when he wrote the original version of nif. See-

[17] G-bang symbols especially shouldn't appear in the expansion itself—that is exactly what we are trying to avoid with gensyms.

[18] We use macroexpand-1 so we only expand the defmacro/g! macro and not the defmacro it expands into.

ing this example use of `defmacro/g!`, we see that no non-gensym bindings will be created in its expansions. `Nif`, defined with `defmacro/g!` like this, is free from variable capture problems.

But since `defmacro/g!` is a macro itself, is it possible that there could be unwanted capture or substitution problems in the macro expansion environment? As with any sufficiently complex abstraction, the behaviour is, to an extent, arbitrary. In the same sense that variable capture itself is a flaw, certain properties of `defmacro/g!` that might appear to be flaws could simply be inherent to its design[19]. As always, the best solution is to understand the abstraction completely.

An interesting *corner-case* of `defmacro/g!` is in G-bang macro defining G-bang macros. All `defmacro/g!` does is introduce a set of bindings into the expansion environment, each of which is bound to a gensym that the macro can use, if it wants. In cases where there are multiple possibilities of where the gensym could be bound, they are always distinguishable because of context. In other words, you can always specify which environment's gensym should be used based on which environment you evaluate it in. Take this contrived example:

```
(defmacro/g! junk-outer ()
  `(defmacro/g! junk-inner ()
     `(let ((,g!abc))
        ,g!abc)))
```

Here there are two gensyms created. The uses of `g!abc` are preceded by only one unquote (comma) so we know that the expansion refers to the inner gensym created by the expansion of `junk-inner`. If each had instead two unquotes, they would refer to the outer gensym created by the expansion of `junk-outer`.

`Defmacro/g!` uses Graham's `flatten` function. `Flatten`, as described in *section 1.3, The Lisp Utility*, takes a tree cons structure—our lisp code—and returns a new list of all the leaves/atoms. The use of `flatten` in `defmacro/g!` is a simple

[19]Though it is never safe to rule out programmer error either.

example of *code-walking*, a topic we will revisit throughout this book.

Exercise: In the above G-bang macro defining G-bang macro, what would be the problem if the first gensym was prefixed with one unquote and the other was prefixed with two?

3.6 Once Only

Peter Norvig is a brilliant programmer and author. His books on AI, especially *Artificial Intelligence: A Modern Approach*[AIMA], are required reading before tackling many of the most difficult problems we currently face as computer scientists. Norvig is perhaps better known to lisp programmers for his book *Paradigms Of Artificial Intelligence Programming: Case Studies in* COMMON LISP[PAIP]. This book is dated but still required reading for serious lisp students and contains many important lisp insights[20]. This section is dedicated to Peter Norvig and is even named after a macro described in PAIP. In its last few pages, tucked away in a description of sequence function implementation, is

> Once-only: A Lesson in Macrology

Which is shortly followed by an even more intriguing sentence:

> [I]f you can understand how to write and when to use once-only, then you truly understand macros.

As we now know, nobody truly understands macros. Understanding a particular macro, even one as important as `once-only`, gets you no further to understanding macros than understanding an important theorem gets you to truly understanding mathematics. Because their possibilities so far seem infinite, truly understanding math or macros is truly impossible.

[20] One bit of COMMON LISP advice from PAIP that is timelessly true is to never mix &optional and &key arguments to lambda or defmacro destructuring forms. This will bite you!

We will not give the definition of Norvig's `once-only` here, but it is a reasonably complex macro with some interesting properties that we will implement slightly differently. `Once-only` was originally written for the deceased *lisp machine* programming environment and was left out of COMMON LISP for inconsequential reasons.

The idea behind `once-only` is to surround a macro expansion with code that will create a new binding. When the macro expansion is evaluated, this binding will be initialised with the result of evaluating one of the forms passed to the macro as an argument. The code in the body of `once-only` can then use the binding which, of course, does not re-evaluate the form that was passed to the macro. The form passed as an argument to the macro is only and always evaluated once. Once-only.

As an example of `once-only`, Norvig shows a `square` macro. Its expansion takes one argument and returns the product of that argument with itself:

```
(defmacro square (x)
  `(* ,x ,x))
```

This will work when we pass a lot of things to it: most variables, numbers, and other forms that can freely be evaluated as many times as necessary. But as soon as forms that have *side-effects* are passed to this version of `square`, all bets are off. The behaviour is, of course, still deterministic, but can be decidedly difficult to determine. With this particular macro, the form will be evaluated exactly twice. But because these things get complicated quickly, in the general case, all bets are off. Making it convenient and easy to avoid these unwanted side-effects is the point of `once-only`. Notice that if we use a function, we get this behaviour for free. After we depart the land of contrived text-book examples, come to our senses, and define `square` as a function, it ends up looking like this:

```
(defun square (x)
  (* x x))
```

Because of how lambda works, we can use any form as the argument to this function definition of `square`. Since this argument will be evaluated exactly once, our notions and conceptual models of side-effects are satisfied. In most instances we expect an expression that we've written only once to be evaluated only once. Conversely, one of the main powers of a macro is to violate this assumption by manipulating the frequency and order of evaluation. In things like loops, for instance, we might want expressions to be evaluated more than once. We might even want them to never be evaluated, say because we want from them something other than their evaluation.

`Once-only` allows us to specify particular parameters in our macro expansion that we would like to only be evaluated once and have their order of evaluation be left-to-right, just like lambda. Here is how we would accomplish this with the traditional `once-only` macro:

```
(defmacro square (x)
  (once-only (x)
    `(* ,x ,x)))
```

But of course if all you ever wanted to do was `once-only` all the arguments of your macro, you would be using a function (lambda) instead. We will return to this point in a moment, but because this book doesn't supply a direct implementation of `once-only`, we introduce an alternate implementation of this functionality for our macro notation. Although there are many interesting implementations of `once-only`[PAIP-P853][PRACTICAL-CL-P95], this section introduces a new technique involving a combination with `defmacro/g!`.

The first step in our once-only implementation is to create some new predicates and utility functions. Again compromising brevity with uniqueness, we reserve another set of symbols for our own use. All symbols starting with the characters `O!` and followed by one or more characters are called *O-bang symbols.*

A predicate to distinguish O-bang symbols from other objects is defined: `o!-symbol-p`. Its definition is nearly identical to that of `g!-symbol-p`. We also introduce a convenient utility function that

Listing 3.7: O-BANG-SYMBOLS

```
(defun o!-symbol-p (s)
  (and (symbolp s)
       (> (length (symbol-name s)) 2)
       (string= (symbol-name s)
                "O!"
                :start1 0
                :end1 2)))

(defun o!-symbol-to-g!-symbol (s)
  (symb "G!"
        (subseq (symbol-name s) 2)))
```

Listing 3.8: DEFMACRO-BANG

```
(defmacro defmacro! (name args &rest body)
  (let* ((os (remove-if-not #'o!-symbol-p args))
         (gs (mapcar #'o!-symbol-to-g!-symbol os)))
    `(defmacro/g! ,name ,args
       `(let ,(mapcar #'list (list ,@gs) (list ,@os))
          ,(progn ,@body)))))
```

changes an O-bang symbol into a G-bang symbol, preserving the characters after the bang: `o!-symbol-to-g!-symbol`. This utility function uses Graham's handy utility function `symb` to create new symbols.

`Defmacro!` represents the final step in our macro defining language—it adds a once-only feature. `Defmacro!` combines with `defmacro/g!` from the previous section. Since `defmacro!` expands directly into a `defmacro/g!` form, it *inherits* the automatic gensym behaviour. Understanding all the pieces being combined is essential for sophisticated combinations. Recall that `defmacro/g!` looks for symbols starting with G-bang and automatically creates gensyms. By expanding into a form with G-bang symbols, `defmacro!` can avoid duplicating gensym behaviour when it implements once-only.

`Defmacro!` gives a shortcut known as *automatic once-only*. With automatic once-only we can prefix one or more of the sym-

bols in the macro's arguments with an O-bang, making them O-bang symbols as defined by `o!-symbol-p`. When we do this, `defmacro!` will know we mean to create a binding in the produced code that will, when evaluated, contain the results of evaluating the code provided as an argument to the macro. This binding will be accessible to the macro expansion through a gensym. But when creating the expansion how can we refer to this gensym? By using the equivalent G-bang symbol as defined above by `o!-symbol-to-g!-symbol`.

The implementation relies on the capabilities of `defmacro/g!`. With the `o!-symbol-to-g!-symbol` utility, we create new G-bang symbols to add into a `defmacro/g!` form. Once we have automatic gensyms, once-only is easy to implement, as evidenced by the brevity of the `defmacro!` definition.

Come back to the land of contrived textbook examples for a moment and we will re-implement the `square` macro, this time with `defmacro!`:

```
(defmacro! square (o!x)
  `(* ,g!x ,g!x))
```

Which we can macroexpand to:

```
* (macroexpand
    '(square (incf x)))

(LET ((#:X1633 (INCF X)))
  (* #:X1633 #:X1633))
T
```

In the previous section I mentioned that we pass a string value to `gensym` for all G-bang symbols. This makes examining the expansions of such forms much easier. Although there is nothing significant about the name of gensyms like `#:X1633`, if we were writing or debugging the `defmacro!` definition of `square` above, we could directly see the connection between this symbol and the symbol used in the macro definition: `X`. Being able to match symbols from definition to expansion and vice-versa is much easier

if this information is preserved in the print-name of the gensyms used, as done in `defmacro/g!` expansions[21].

Aside from the less verbose usage and more helpful expansion output compared to the traditional `once-only`, `defmacro!` also provides one extra key feature. In the traditional `once-only`, the binding for the gensym used to access the created lexical variable is given the same name as the argument to the macro expansion, which *shadows* the macro argument so it cannot be accessed by the macro definition. Because `defmacro!` splits this into two separate types of symbols, G-bang symbols and O-bang symbols, we can write macro expansions that use both of these values. To demonstrate this, here is yet another definition of the `square` macro:

```
(defmacro! square (o!x)
  `(progn
     (format t "[~a gave ~a]~%"
                 ',o!x   ,g!x)
     (* ,g!x ,g!x)))
```

Which can be used like so:

```
* (defvar x 4)

X
* (square (incf x))
[(INCF X) gave 5]
25
```

Notice that we *quote* the unquoted O-bang symbol in the above `square` definition. We do this because we don't want to evaluate this form again. The expansion generated by `defmacro!` already evaluated it. We simply want to take the form passed to square and use it for another purpose, in this case some kind of crude debugging statement. However, even though we evaluated

[21] This is also the reason for the number in the print-name of a gensym, specified by `*gensym-counter*`. This counter lets us distinguish instances of gensyms with the same print name.

Listing 3.9: NIF

```
(defmacro! nif (o!expr pos zero neg)
  `(cond ((plusp ,g!expr) ,pos)
         ((zerop ,g!expr) ,zero)
          (t ,neg)))
```

it once already, and in this case it being incorrect, there is nothing stopping us from evaluating the provided form again, should our desired abstraction demand it.

The `defmacro!` language allows us granular, convenient control over the evaluation of the arguments passed to our macros. If we prefix all the symbols representing arguments in the macro definition with O-bang, and only use the corresponding G-bang symbols in the macro definition, our expansions will be the same as lambda expressions—each form evaluated once, in left to right order. Without any of these symbols in `args` and without using any G-bang symbols in the expansion, `defmacro!` acts just like the regular COMMON LISP `defmacro`.

`Defmacro!` is most useful during iterative development of a macro. Because it is a simple matter of adding two characters to a macro argument to get lambda style evaluation, and using gensyms is as easy as, well, writing them, we can change our mind about these decisions instantly. `Defmacro!` feels like an even tighter fitting glove over lambda than does COMMON LISP's `defmacro`. It is for this reason, iterative development, that we will use `defmacro!` as the main macro definition interface for the remainder of this book.

Let's return to Graham's `nif` macro. When updating this macro for `defmacro!`, we notice that the `expr` argument, the one for which we created a gensym, is evaluated exactly once. Here we use `defmacro!` to indicate that this argument should be evaluated only once by calling it `o!expr`. This implementation of `nif` represents the final step in our evolution of this macro.

`Defmacro!` blurs the gap between macro and function. It is this feature, the ability to provide some O-bang symbols in the macro argument and some regular symbols, that makes

`defmacro!` especially useful. Just as backquote allows you to flip the default quoting behaviour, `defmacro!` allows you to flip the evaluation semantics of macro arguments from regular unevaluated macro forms to singly evaluated, left-to-right lambda arguments.

3.7 Duality of Syntax

One of the most important concepts of lisp is called *duality of syntax*. Understanding how to use dualities and why they are important is an underlying theme of macro writing and of this book. Dualities are sometimes designed and sometimes accidentally discovered. To programmers of non-lisp languages the reality of dual syntax would be too unbelievable to describe at this point in the book so we will for now shy away from a direct definition. Instead, you, the reader, gets to discover it again and again as it is applied slowly and carefully so as to avoid shock. Should you experience headaches or other discomfort through the course of this book, I recommend that you immediately execute a garbage collection cycle (get some sleep), then return with a fresh and open mind.

Referential transparency is sometimes defined as a property of code where any expression can be inserted anywhere and always have the same meaning. Introducing syntactic duals is the conscious violation of referential transparency and discovering them is reaping the fruits of a language that enables such violations. While other languages only let you build with semi-transparent panes of glass, lisp lets you use an assortment of smoke, mirrors, and prisms. The magic dust is made of macros, and most of its best tricks are based on syntactic duals.

This section describes an important dual syntax we have already discussed but have not yet completely explored: COMMON LISP uses the same syntax for accessing both of its major types of variables, dynamic and lexical. This book tries to illustrate the real power of dynamic and lexical scope and why COMMON LISP's decision to use dual syntax is important.

The purpose of dynamic scope is to provide a way for getting values in and out of lisp expressions based on when the expression

is evaluated, not where it is defined or compiled. It just so happens that, thankfully, the syntax that COMMON LISP defines for this is identical to that used to access lexical variables, which are the exact opposite of dynamic variables in that they always refer to the locations they were compiled for, independent of when the access takes place. In fact, without external context in the form of a declaration, you can't tell which type of variable an expression is referring to. This dual syntax violates referential transparency, but rather than being something to avoid, lisp programmers welcome this because just as you can't differentiate an expression without context, neither can a macro. Hold that thought for a second. First, it must be made clear that creating bindings for dynamic variables does not create lexical closures. As an example, let's re-bind the variable `temp-special` that we earlier declared special:

```
* (let ((temp-special 'whatever))
    (lambda () temp-special))

#<Interpreted Function>
```

Even though it is a *let over lambda*, this is not a lexical closure. This is a simple evaluation of a `lambda` macro form in some dynamic context which results in, of course, an anonymous function. This function, when applied, will access whatever current dynamic environment exists and fetch that value of `temp-special`. When the `lambda` macro was evaluated, a dynamic binding of `temp-special` to the symbol `whatever` existed, but who cares? Remember that `lambda` forms are constant objects, just simple machine code pointer returners, so evaluating this lambda form never even accesses the dynamic environment. What happens to our symbol `whatever`? After lisp is done evaluating the lambda form, it removes it from the dynamic environment and throws it away, unused.

Some early lisps did support *dynamic closures*, which meant that every function defined in a non-null dynamic environment had its own (possibly partially shared) stack of dynamic bindings. The effect is similar to COMMON LISP's lexical scope and was

implemented with something termed a *spaghetti stack*[SPAGHETTI-STACKS][INTERLISP-TOPS20]. This data structure is no longer a stack data structure, but actually a multiple path, garbage collected network. COMMON LISP does away with spaghetti stacks and only provides lexical closures[MACARONI].

So lexical and dynamic variables are actually completely different, deservedly distinct concepts that just happen to share the same syntax in COMMON LISP code. Why on earth would we want this so-called duality of syntax? The answer is subtle, and only consciously appreciated by a minority of lisp programmers, but is so fundamental that it merits close study. This dual syntax allows us to a write a macro that has a single, common interface for creating expansions that are useful in both dynamic and lexical contexts. Even though the meanings of expansions of the macro can be completely different given their context, and even though each can mean entirely different things underneath, we can still use the same macro and the same combinations of this macro with other macros. In other words, macros can be made *ambivalent* about not only the contents of their macro arguments, but also about the different meanings of their expansions. We can use the macro just for its understood code transformation, ignoring the semantic meanings of the code, all because the code only has meaning once we use it somewhere—it has no meaning during macro processing. The more dualities of syntax there are, the more powerful an associated macro becomes. Many more examples of the advantages of dual syntax are detailed through this book. The duality between dynamic and lexical variables is a mild (but useful) example of this lispy philosophy. Some macros are created for the specific purpose of having powerful duals, and sometimes there are many more than two possible meanings for an expansion.

A traditional convention in COMMON LISP code is to prefix and postfix the names of special variables with asterisk characters. For example, we might've chosen to name our `temp-special` variable `*temp-special*`. Since this convention is almost like having another namespace for dynamic variables, diminishing their duality with lexical variables, this book does not follow it exactly.

The asterisks are merely convention and, fortunately, COMMON LISP does not enforce them. Not only can we leave the asterisks off special variable names, but we can add them to lexical variable names. Maybe it is a question of style. Which is a lesser fashion crime: lexical variables with asterisks or special variables without? I tend to think the less verbose of the two. Also, the names of lexical and special variables can be gensyms, a concept that transcends print names on symbols.

So, as mentioned, this book hijacks the usual asterisk convention. Instead of

> Asterisked variable names indicate special variables.

this book uses

> Asterisked variable names indicate special variables defined by the standard.

My largest motivation for dropping these variable name earmuffs is simple and subjective: I think they are annoying to type and make code look ugly. I will not go so far as to suggest you do this for your own programs, just mention that I have been leaving off the earmuffs for years and am very content with COMMON LISP.

Chapter 4

Read Macros

4.1 Run-Time at Read-Time

> Syntactic sugar causes cancer of the semicolon.
>
> —Alan Perlis

Not only does lisp provide direct access to code that has been parsed into a cons cell structure, but it also provides access to the characters that make up your programs before they even reach that stage. Although regular macros work on programs in the form of trees, a special type of macro, called a *read macro*, operates on the raw characters that make up your program.

In lisp, when we want to define a non-lisp syntax, it doesn't make sense to use the lisp reader—that is only for reading lisp. The read macro is the device we use to process non-lisp syntax before the lisp reader gets its paws on it. The reason why the lisp reader is more powerful than that of other languages is that lisp gives you *hooks* for controlling every aspect of its behaviour. In particular, lisp lets you *extend* the reader, so that non-lisp objects are actually read in as lisp objects. Just as you build your applications on top of lisp, extending it with macros and functions, lisp applications can, and frequently do, ooze out into this dimension of extensibility as well. When this happens, any character based syntax can be read with the lisp reader, meaning that you have turned the syntax into lisp.

While the transformations of code done by regular macros are only used for turning lisp code into new lisp code, read macros can be created so as to turn non-lisp code into lisp code. Like regular macros, read macros are implemented with functions underneath so we have available the full power of the lisp environment. Like macros that increase productivity because they create more concise domain specific languages for the programmer to use, read macros boost productivity by allowing expressions to be abbreviated to the point where they aren't even lisp expressions anymore. Or are they?

If all we have to do to parse these non-lisp domain specific languages is write a short read macro, maybe these non-lisp languages really are lisp, just in a clever disguise. If XML can be directly read in by the lisp reader[XML-AS-READ-MACRO], maybe XML, in a twisted sort of sense, is actually lisp. Similarly, read macros can be used to read regular expressions and SQL queries directly into lisp, so maybe these languages really are lisp too. This fuzzy distinction between code and data, lisp and non-lisp, is the source of many interesting philosophical issues that have perplexed lisp programmers since the very beginning.

A basic read macro that comes built in with COMMON LISP is the #. read-time eval macro. This read macro lets you embed objects into the forms you read that can't be serialised, but can be created with a bit of lisp code. One fun example is making forms that become different values each time they are read:

```
* '(football-game
    (game-started-at
      #.(get-internal-real-time))
    (coin-flip
      #.(if (zerop (random 2)) 'heads 'tails)))

(FOOTBALL-GAME
  (GAME-STARTED-AT 187)
  (COIN-FLIP HEADS))
```

Even though it is the same expression, this form reads in differently each time:

```
* '(football-game
     (game-started-at
       #.(get-internal-real-time))
     (coin-flip
       #.(if (zerop (random 2)) 'heads 'tails)))

(FOOTBALL-GAME
  (GAME-STARTED-AT 309)
  (COIN-FLIP TAILS))
```

Note that the two forms surrounded by #. are evaluated at read time, not when the form is evaluated. The complete list has been formed after they are evaluated, and the before-after equivalence (as defined by `equal`) can be seen by re-evaluating the last form read in and comparing it with the previous results, using the * and + convenience variables of the REPL[1]:

```
* (equal * (eval +))

T
```

Notice that because these forms are actually evaluated at read-time, this is different from using backquote, which we will look at more closely in the following section. We can evaluate a similar form that uses backquotes:

```
* `(football-game
     (game-started-at
       ,(get-internal-real-time))
     (coin-flip
       ,(if (zerop (random 2)) 'heads 'tails)))

(FOOTBALL-GAME
  (GAME-STARTED-AT 791)
  (COIN-FLIP HEADS))
```

[1]The * variable contains the value that resulted from evaluating the previous form, and the + variable contains that form.

but it will evaluate to a different result when we re-evaluate it, since backquote reads in as code for evaluation:

```
* (equal * (eval +))

NIL ; unless you're really fast and lucky
```

4.2 Backquote

Backquote, sometimes known as *quasiquote*[2], and displayed as `, is a relative new-comer to mainstream lisp programming, and the concept is still almost completely foreign to languages other than lisp.

Backquote has a bizarre history of development in parallel with lisp. It is reported[QUASIQUOTATION] that early on nobody believed that nested backquotes worked right until a sharp programmer realised that they actually did work right—people's ideas of what was right were wrong. The nested backquote is notoriously difficult to understand. Even Steele, the father of COMMON LISP, complains about it[CLTL2-P530].

In principle, lisp doesn't need backquote. Anything that can be done with backquote can be done with other list building functions. However, backquote is so useful for macro programming, which in lisp means all programming, that lisp professionals have come to rely on it heavily.

First off, we need to understand regular quotation. In lisp, when we prefix a form with the quote character (') we are informing lisp that the following form should be treated as raw data, and not code to be evaluated. Rather, quote reads in as code that, when evaluated, returns a form. We sometimes say that quote *stops* or *turns off* the evaluation of a form.

Backquote can be used as a substitute for quote in lisp. Unless certain special characters, called *unquote* characters, appear in a form, backquote stops evaluation in the same way as quote. As the name suggests, these unquote characters reverse the evaluation

[2]Scheme programmers call it quasiquote and COMMON LISP programmers call it backquote.

semantics. We sometimes say that an unquote *restarts* or *turns back on* the evaluation of a form.

There are three main types of unquote: regular unquote, splicing unquote, and destructive splicing unquote.

To perform a regular unquote, we use the comma operator:

```
* (let ((s 'hello))
    `(,s world))

(HELLO WORLD)
```

Although the expression we are unquoting is simply a symbol to evaluate, s, it can instead be any lisp expression that evaluates to something meaningful for whatever context it appears in the backquote template. Whatever the results are, they are inserted into the resulting list in the car position of where they appear in the backquote template.

In lisp form notation, we can use . to indicate that we want to explicitly put something in the cdr of the list structure we are creating. If we put a list there, the resulting form from the backquote will remain a valid list. But if we put something else there, we will get a new, non-list structure.

We have this ability inside of backquote like everywhere else[3]. Thanks to the design of backquote, we can even unquote things in this position:

```
* (let ((s '(b c d)))
    `(a . ,s))

(A B C D)
```

Inserting lists into the cdr position of a list being created from a backquote template is so common that backquote takes it a step further with splicing unquote. The ., combination above is useful, but is incapable of inserting elements into the middle of a list. For that, we have the splicing unquote operator:

[3]Because backquote uses the standard `read` function like (nearly) everywhere else.

```
* (let ((s '(b c d)))
    `(a ,@s e))

(A B C D E)
```

Neither ., nor ,@ modify the list being spliced in. For instance, after evaluating the backquote in both of the previous forms, s will still be bound to the three element list (B C D). Although it is not strictly required to by the standard, the (B C D) in the (A B C D) form above is allowed to share structure with the spliced-in list, s. However, in the list (A B C D E), this list structure is guaranteed to be freshly allocated when the backquote is evaluated since ,@ is forbidden to modify the lists being spliced in. Splicing unquote is non-destructive because generally we want to think about backquote as being a re-usable template for creating lists. Destructively modifying the list structure of data that isn't freshly allocated on every evaluation of the backquote code can have undesirable effects upon future expansions.

However, COMMON LISP also provides a destructive version of splicing unquote which can be used anywhere splicing unquote can. To splice destructively, use ,. instead. Destructive splicing works the same as regular splicing except that the list being spliced in may be modified during the evaluation of the backquote template. As well as being only one character different from regular splicing, this notation is a clever re-use of the . character from the ., cdr position unquoting we looked at above.

To see this in action, here we destructively modify the list pointed to by to-splice:

```
* (defvar to-splice '(B C D))

TO-SPLICE
* `(A ,.to-splice E)

(A B C D E)
* to-splice

(B C D E)
```

Destructively modifying lists to be spliced in can be dangerous. Consider the following use of destructive splicing:

```
(defun dangerous-use-of-bq ()
  `(a ,.'(b c d) e))
```

The first time dangerous-use-of-bq is called, the expected answer is returned: (A B C D E). But since it uses destructive splicing and modifies a list that isn't freshly generated—the quoted list—we can expect various undesirable consequences. In this case, the second time dangerous-use-of-bq is evaluated, the (B C D) form is now really a (B C D E) form, and when backquote tries to destructively splice this list onto the remainder of the backquote template, (E)—its own tail—it creates a list containing a *cycle*. We discuss cycles in more detail in *section 4.5, Cyclic Expressions.*

However, there are many cases where destructive splicing is perfectly safe. Don't let dangerous-use-of-bq scare you if you need more efficiency in your backquote forms. There are many operations that create fresh list structure that you are probably planning on throwing out anyways. For instance, splicing the results of a mapcar is so common and safe that the following could probably become a programming idiom:

```
(defun safer-use-of-bq ()
  `(a
    ,.(mapcar #'identity '(b c d))
    e))
```

But there is a reason it hasn't. The most common use of backquote is for authoring macros, the part of lisp programming where speed matters least and where clarity matters most. If thinking about the *side-effects* of your splicing operations distracts you even for a split second while creating and interpreting macros, it is probably not worth the trouble. This book sticks with regular splicing. The most common use of backquote is in macro construction but this is not its only use. Backquote is actually

a useful domain specific language for the domain of mashing together lists, one made even more useful given the possibility of destructive splicing.

How does backquote work? Backquote is a read macro. Backquoted forms read in as code that, when evaluated, becomes the desired list. Going back to the example of the previous section on read-time evaluation, we can turn off *pretty printing*, quote the value of the backquote form, and print it out to see exactly how backquote forms read[4]:

```
* (let (*print-pretty*) ; bind to nil
    (print
      ''(football-game
          (game-started-at
            ,(get-internal-real-time))
          (coin-flip
            ,(if (zerop (random 2))
               'heads
               'tails))))
    t)

(LISP::BACKQ-LIST
  (QUOTE FOOTBALL-GAME)
  (LISP::BACKQ-LIST
    (QUOTE GAME-STARTED-AT)
    (GET-INTERNAL-REAL-TIME))
  (LISP::BACKQ-LIST
    (QUOTE COIN-FLIP)
    (IF (ZEROP (RANDOM 2))
      (QUOTE HEADS)
      (QUOTE TAILS))))
T
```

In the above *ugly printed* form, the function `LISP::BACKQ-LIST` is identical to list, except for its pretty printing behaviour.

[4]We return `t` so we don't see the value returned from `print`. `(values)` is also common.

Notice that the comma operators are gone. COMMON LISP is fairly liberal in what it allows backquote to read in as, particularly for operations where shared structure is permitted.

Backquote also provides many interesting solutions to the amusing *non-problem* of writing a lisp expression that evaluates to itself. These expressions are commonly called *quines* after Willard Quine who studied them extensively and who, in fact, coined the term quasiquote—an alternative name for backquote[FOUNDATIONS-P31-FOOTNOTE3]. Here is a fun example of a quine that is attributed to Mike McMahon in [QUASIQUOTATION]:

```
* (let ((let '`(let ((let ',let))
                  ,let)))
    `(let ((let ',let)) ,let))

(LET ((LET '`(LET ((LET ',LET))
                ,LET)))
  `(LET ((LET ',LET)) ,LET))
```

To save you the *mental code-walk*:

```
* (equal * +)

T
```

Exercise: In the following evaluation, why is the backquote expanded into a regular quote? Isn't it quoted?

```
* '`q

'Q
```

4.3 Reading Strings

In lisp, strings are delimited by the double-quote (") character. Although strings can contain any character in the character set of your lisp implementation, you can't directly insert certain characters into the string. If you want to insert the " character or the \

Listing 4.1: SHARP-DOUBLE-QUOTE

```
(defun |#"-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let (chars)
    (do ((prev (read-char stream) curr)
         (curr (read-char stream) (read-char stream)))
        ((and (char= prev #\") (char= curr #\#)))
      (push prev chars))
    (coerce (nreverse chars) 'string)))

(set-dispatch-macro-character
  #\# #\" #'|#"-reader|)
```

character, you will need to prefix them with \ characters. This is called *escaping* the characters. For example, here is how to enter a string containing " and \ characters:

```
* "Contains \" and \\."

"Contains \" and \\."
```

This obviously works, but sometimes typing these \ characters becomes tedious and error prone. This is lisp, of course, and if we don't like something we are free, even encouraged, to change it. In this spirit, this book presents a read macro called #", or sharp-double-quote. This read macro is for creating strings containing " and \ characters without invoking escapes.

Sharp-double-quote[5] will start reading the string immediately after its invocation characters: # and ". It will continue to read, character by character, until it encounters the two characters " and #, in sequence. When it finds this terminating sequence, it returns the string represented by all the characters between the #" and "#. The sharp-double-quote read macro used to be used for bit-strings, but COMMON LISP freed up this useful macro

[5]Our convention of naming the underlying functions of read macros with a symbol based on the read macro's characters, like \#\verb"—reader—, is due to Steele in CLtL2.

character for us by moving bit-strings to the #* read macro[EARLY-CL-VOTES].

Here is an example evaluation of our new sharp-double-quote:

```
* #"Contains " and \."#

"Contains \" and \\."
```

Notice that when the REPL prints the string, it still uses the " character as a delimiter, so the " and \ characters are still escaped in the printed representation of the string. These strings are simply read in as if you escaped the characters manually.

But sometimes #" isn't good enough. For instance, in this U-language paragraph you are reading right now, I have included the following sequence of characters: "#. Because of this, this paragraph couldn't have been delimited with #" and "#. And because I hate escaping things, take my word that it wasn't delimited with regular double quotes.

We need a read macro that allows us to customise the delimiter for each specific context where we use it. As is often the case, we need to look no further than Larry Wall's Perl language for inspiration on the design of programming shortcuts. Perl is a beautiful, wonderfully designed language and possesses many great ideas that are ripe for *pilfering* by lisp. Lisp is, in some sense, a big blob, a snowball perhaps, that rolls around assimilating ideas from other programming languages, making them its own[6].

The #> read macro is directly inspired by Perl's << operator. This operator allows Perl programmers to specify a string of text to act as the end delimiter for a quoted string. #> reads characters until it finds a newline character, then reads characters, one-by-one, until it encounters a sequence of characters identical to the characters it found immediately after #> and before the newline.

For example:

```
* #>END
```

[6]The most cited example of this is objects, but there are countless other examples such as the `format` function from FORTRAN.

Listing 4.2: SHARP-GREATER-THAN

```
(defun |#>-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let (chars)
    (do ((curr (read-char stream)
               (read-char stream)))
        ((char= #\newline curr))
      (push curr chars))
    (let* ((pattern (nreverse chars))
           (pointer pattern)
           (output))
      (do ((curr (read-char stream)
                 (read-char stream)))
          ((null pointer))
        (push curr output)
        (setf pointer
              (if (char= (car pointer) curr)
                (cdr pointer)
                pattern))
        (if (null pointer)
          (return)))
      (coerce
        (nreverse
          (nthcdr (length pattern) output))
        'string))))

(set-dispatch-macro-character
  #\# #\> #'|#>-reader|)
```

```
I can put anything here: ", \, "#, and ># are
no problem. The only thing that will terminate
the reading of this string is...END

"I can put anything here: \", \\, \"#, and ># are
no problem. The only thing that will terminate
the reading of this string is..."
```

4.4 CL-PPCRE

CL-PPCRE[CL-PPCRE] is a high-performance regular expression library written on COMMON LISP. It was created by the widely respected lisp hacker Edi Weitz. On behalf of the lisp professionals everywhere who have benefited so much from CL-PPCRE and his other software, this section is dedicated to Edi Weitz. When other people are talking, Edi is coding; code speaks louder than argument.

PPCRE, for those who aren't already familiar, stands for Portable Perl Compatible Regular Expressions. CL-PPCRE, like the code in this book, is *portable* because it can run in any ANSI-compliant COMMON LISP environment. CL-PPCRE, also like the code in this book, is open-source and freely available. Although CL-PPCRE is almost perfectly compatible with Perl, it is different from Perl in a few important ways. CL-PPCRE provides several notable lispy enhancements to regular expressions. There are three substantial ways that CL-PPCRE is different from the implementation of regular expressions in Perl.

First, CL-PPCRE is fast. Really fast. When compiled with a good native code compiler, benchmarks suggest that for most regular expressions CL-PPCRE is roughly twice as fast as Perl, often much faster. And Perl has one of the fastest non-lisp regular expression engines around: a highly optimised engine written in C. How is this possible? Surely Perl's low-level implementation should have a performance edge over anything written in a high-level language like lisp.

This misconception is known as the *performance myth*, the general version of which is the following: low level languages result

in faster code because you can program closer to the hardware. As this book hopes to explain, for complicated systems this myth is false. Examples like CL-PPCRE demonstrate this. The more low-level a language is, the more it prevents you and your compiler from making the efficiency optimisations that actually matter.

With CL-PPCRE, the technical reason for the performance boost is simple: COMMON LISP, the language used to implement CL-PPCRE, is a more powerful language than C, the language used to implement Perl. When Perl reads in a regular expression, it can perform analysis and optimisation but eventually the regular expression will be stored into some sort of C data structure for the static regular expression engine to use when it attempts the matching. But in COMMON LISP—the most powerful language—it is essentially no more difficult to take this regular expression, convert it into a lisp program, and pass that lisp program to the optimising, native-code lisp compiler used to build the rest of your lisp system[7]. Because programs compiled with a C compiler don't have access to the C compiler, Perl is unable to compile regular expressions all the way down to machine code. Lisp's compilation model is in a different class from C altogether. In COMMON LISP, compiling things at run-time (as at anytime) is portable, seamless, done in the same process as your lisp image, garbage collected when no longer needed, and, due to its incremental nature, highly efficient.

The second major difference between CL-PPCRE and Perl is that CL-PPCRE isn't tied to a string-based notation for regular expressions. CL-PPCRE has been freed from a character representation and permits us to encode regular expressions as lisp forms (sometimes called *S-expressions*). Since such forms are the very notation we use for writing lisp programs and macros, we are allowed many more opportunities for *cohesion* in our abstractions. See the documentation and code of CL-PPCRE[CL-PPCRE] for details on using this regular expression notation, and also for

[7]CL-PPCRE is actually more sophisticated than described here. It has its own compilation function and usually (unless you're building regular expressions at run-time) ensures that this is called for you when your lisp program is compiled.

Listing 4.3: SEGMENT-READER

```
(defun segment-reader (stream ch n)
  (if (> n 0)
    (let ((chars))
      (do ((curr (read-char stream)
                 (read-char stream)))
          ((char= ch curr))
        (push curr chars))
      (cons (coerce (nreverse chars) 'string)
            (segment-reader stream ch (- n 1))))))
```

an example of a well-designed, lispy domain specific language.

Sure, CL-PPCRE is great, but why are we discussing it in a chapter about read macros? The answer lies in the third and last way that CL-PPCRE is substantially different from Perl. In Perl, regular expressions are closely tied into the language. While lisp's syntax is the way it is to accommodate meta-programming, Perl's syntax is the way it is to accommodate regular expressions and other sorts of syntactic shortcuts. Part of the reason we use regular expressions so often in Perl code is due to the experience of writing them being so brief and painless.

To add a convenient programmer interface in a Perlish style, read macros come in very handy. Because programming read macros is programming lisp, we start off with a utility function: `segment-reader`. Given a stream, a delimiter character, and a count, `segment-reader` will read characters from the stream until the delimiter character is encountered. If the count is greater than 1, `segment-reader` will return a cons. The car of this cons is a string and the cdr is the result of a recursive invocation of `segment-reader` given a decremented count parameter to get the next segment[8].

For example, reading 3 segments from the stream t[9] with a

[8] In COMMON LISP, when the test clause of an if form missing the else clause is found false, `nil` is returned from the if. Experienced COMMON LISP programmers often rely on this behaviour, as we do here in `segment-reader` for the base case of a recursion building up a list.

[9] The stream `t` corresponds to standard input when done from the REPL

delimiter character of / is done like so:

```
* (segment-reader t #\/ 3)
abc/def/ghi/

("abc" "def" "ghi")
```

Perl programmers will probably see exactly where this is going. The idea is, with full apologies to Larry Wall, to *pilfer* the syntax for two handy Perl regular expression operators. In Perl, if we want to try matching a regular expression to a variable, we can write

```
$my_boolean = ($var =~ m/^\w+/);
```

to see if the contents of `$var` begin with one or more alphanumeric characters. Similarly, if we want to apply a *substitution* regular expression, we can also use the Perl =~ operator to apply a substitution regular expression to change the first occurrence of dog to cat in our string variable `$var`:

```
$var =~ s/dog/cat/;
```

The great thing about the Perl syntax is that the delimiter character can be any character that is convenient for the programmer. If we wanted to use a regular expression or a substitution containing the / character, we could use a different character to avoid any conflicts[10]:

```
$var =~ s|/usr/bin/rsh|/usr/bin/ssh|;
```

Defining a read macro to copy these two Perl syntaxes gives us a chance to demonstrate an interesting macro technique, the double backquote. The idea is that sometimes, as in the `match-mode-ppcre-lambda-form` and `subst-mode-ppcre-lambda-form` macros,

here.

[10]This may not be due to Perl; TeX's verbatim quotations provide something similar.

Listing 4.4: MATCH-MODES

```
#+cl-ppcre
(defmacro! match-mode-ppcre-lambda-form (o!args)
 ``(lambda (,',g!str)
     (cl-ppcre:scan
       ,(car ,g!args)
       ,',g!str)))

#+cl-ppcre
(defmacro! subst-mode-ppcre-lambda-form (o!args)
 ``(lambda (,',g!str)
     (cl-ppcre:regex-replace-all
       ,(car ,g!args)
       ,',g!str
       ,(cadr ,g!args))))
```

we want to write code that generates lists. Notice that when you normally define a macro and use a single backquote, you are generating a list representing code and returning it from the macro for it to be spliced into expressions for evaluation. With a double backquote you are still generating a list representing code, but this code will, when evaluated, itself use code built by a backquote in order to return a list. In our case, these two macros expand into code that you can evaluate to create lambda forms that are useful for applying CL-PPCRE regular expressions.

We prefix these macros, and some other expressions below, with a #+ read macro. This read macro tests whether we have CL-PPCRE available[11] before evaluating the following form. If CL-PPCRE isn't available when loading the source code from this book, the functionality of this section will not be available.

Finally, we can define a reader function to pull together these utilities then add this function to our macro dispatch table. We chose to use the #~ read macro because it is a nice analog to the Perl =~, the source of inspiration for our syntax.

The #~ read macro is designed to be convenient. Here is how

[11] It tests for CL-PPCRE by searching for the presence of the keyword symbol `:CL-PPCRE` in the list stored in the `*features*` variable.

Listing 4.5: CL-PPCRE-READER

```
#+cl-ppcre
(defun |#~-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let ((mode-char (read-char stream)))
    (cond
      ((char= mode-char #\m)
         (match-mode-ppcre-lambda-form
           (segment-reader stream
                           (read-char stream)
                           1)))
      ((char= mode-char #\s)
         (subst-mode-ppcre-lambda-form
           (segment-reader stream
                           (read-char stream)
                           2)))
      (t (error "Unknown #~~ mode character")))))

#+cl-ppcre
(set-dispatch-macro-character #\# #\~ #'|#~-reader|)
```

we can create a regular expression matching function:

```
* #~m/abc/

#<Interpreted Function>
```

We can now apply this function to a string just as a normal function call[12]:

```
* (funcall * "123abc")

3
6
#()
#()
```

[12]The * variable is bound to the value returned from the evaluation of the last expression entered in the REPL. Here it is bound to our regular expression function.

The values returned are from the `cl-ppcre:scan` function, documentation for which can be found in [CL-PPCRE]. If you are only interested in whether the string matched, the fact that the first value returned is not `nil` means that it did. Generalised booleans, and why they are an important feature of COMMON LISP, are discussed further in *chapter 6, Anaphoric Macros.*

We can also create substitution regular expression functions. A slight difference between Perl and our read macro is that substitution regular expression functions do not modify their arguments. They will return new strings, which are copies of the original strings with the substitutions made. Another difference is that, by default, this read macro substitutes all occurrences of the pattern instead of just the first in the string. In Perl you need to add a global modifier to your regular expression to get this behaviour, but not here:

```
* (funcall #~s/abc/def/ "Testing abc testing abc")

"Testing def testing def"
```

So how does this work? What do #~ expressions, which are clearly not lisp expressions, read in as? On the surface, it appears as though they read in as functions, but this turns out to not be the case. Let's quote one of these forms so we can see what it is according to the lisp reader:

```
* '#~m|\w+tp://|

(LAMBDA (#:STR1)
  (CL-PPCRE:SCAN "\\w+tp://" #:STR1))
```

Substitutions are similar:

```
* '#~s/abc/def/

(LAMBDA (#:STR2)
  (CL-PPCRE:REGEX-REPLACE-ALL
    "abc"
```

```
    #:STR2
    "def"))
```

They are read in as lambda forms. So as far as the lisp reader is concerned, we didn't write it in some funny non-lisp language after all. This is a function designator. Since our expressions are simply lists with the first element the symbol `lambda`, recall from *section 2.4, Let It Be Lambda* how we can use lambda forms in the first argument of a function call to invoke anonymous functions:

```
* (if (#~m/^[\w-.]+$/ "hcsw.org")
    'kinda-looks-like-a-domain
    'no-chance!)

KINDA-LOOKS-LIKE-A-DOMAIN
```

When we use `funcall` or `apply` to use the objects read in by `#~`, we make use of the ANSI `lambda` macro but not when the form is the first in the list: a useful *duality of syntax*. If our `#~` expressions read in as sharp-quoted lambda forms we wouldn't be able to use them in the function position of an expression—only function names and lambda forms can go there. So for both tasks there only needs to be one read macro, which is fortunate because it is a large and complicated one. Taking advantage of dual syntax lets us focus on getting the correct expansion instead of tracking different syntax requirements. Instead of one interesting macro, we got two. To save effort, make your syntax as similar as possible.

A common problem when using CL-PPCRE is to forget to *escape* backslashes in your regular expressions. Look what happens when you do this:

```
* "\w+"

"w+"
```

This is a string of length 2. Where did the backslash go? Double-quote thought we meant to escape the w character instead of writing a literal \ character. For our `#~` read macro that just

reads characters and looks for the appropriate delimiter, this is not an issue and we can write regular expressions just as we do in Perl—without escapes. See the quoting of the URL regular expression above.

Although the #~ read macro defined in this section is already very convenient, there is still room for improvement and enhancement. Exercise: Improve it. The most obvious first step is to support regular expression modifiers, such as case insensitivity in matches. If done with the same syntax as Perl, this will involve using the function `unread-char`, which is common in read macros to avoid accidentally *eating* a character that some other read macro might be expecting.

4.5 Cyclic Expressions

All our talk about lisp programs being *trees* of cons cells has actually been a small lie. Sorry about that. Lisp programs are actually not trees but are instead *directed acyclic graphs*—trees with possibly shared branches. Since the evaluator doesn't care about where the branches it is evaluating come from, there is nothing wrong with evaluating code with shared structure.

A useful provided read macro is #=. We already saw how lisp can be made to output forms with the #= macro when serialising macro expansions in *section 3.5, Unwanted Capture.* #= and its partner ## let you create self-referential S-expressions. This allows you to do things like represent shared branches in directed acyclic graphs and other interesting data structures with little or no effort.

But most importantly it allows you to serialise data without having to disassemble and reassemble an efficient in-memory data structure where large portions of the data are shared. Here is an example where the two lisp lists read in are distinct objects (not eq):

```
* (defvar not-shared '((1) (1)))

((1) (1))
```

```
* (eq (car not-shared) (cadr not-shared))

NIL
```

But in the following example, with serialised data using the #= read macro, the 2 lists really are the same list:

```
* (defvar shared '(#1=(1) #1#))

((1) (1))
* (eq (car shared) (cadr shared))

T
```

As mentioned, we can give shared, acyclic list structure to the evaluator with no trouble:

```
* (list
    #1=(list 0)
    #1#
    #1#)

((0) (0) (0))
```

If we print the last form we just evaluated, we see it the same way the lisp evaluator does: a regular list with three separate branches:

```
* +

(LIST (LIST 0) (LIST 0) (LIST 0))
```

But if we bind the `*print-circle*` special variable to a non-nil value when we print it, we see that the expression is not really a tree at all, but instead a directed acyclic graph:

```
* (let ((*print-circle* t))
    (print ++)
```

```
    t)

(LIST #1=(LIST 0) #1# #1#)
T
```

As another fun example, here's how to print an infinite list by pointing the cdr of a cons to itself, forming a so-called *cycle* or *circle*:

```
* (print '#1=(hello . #1#))

(HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 ...
```

So unless you want that to happen, be sure you set `*print-circle*` when *serialising* cyclic data structures:

```
* (let ((*print-circle* t))
    (print '#1=(hello . #1#))
    nil)

#1=(HELLO . #1#)
NIL
```

Is there an easy way to tell if a portion of list structure is cyclic or contains shared structure? Yes, the provided `cyclic-p` predicate uses the most obvious algorithm for discovering this: recurse across the structure keeping a *hash-table* up to date with all the cons cells you've encountered so far. If you ever come across a cons cell that already exists in your hash-table, you have already been there and therefore have detected a cycle or a shared branch. Notice that because it only recurses across cons cells, `cyclic-p` is incapable of discovering such references in data structures like vectors.

Finally, because most (see [SYNTACTICALLY-RECURSIVE]) lisp compilers forbid you from passing circular forms to the compiler, executing the following is undefined but likely to break your compiler by putting it into an infinite compiling loop:

Listing 4.6: CYCLIC-P

```
(defun cyclic-p (l)
  (cyclic-p-aux l (make-hash-table)))

(defun cyclic-p-aux (l seen)
  (if (consp l)
    (or (gethash l seen)
        (progn
          (setf (gethash l seen) t)
          (or (cyclic-p-aux (car l) seen)
              (cyclic-p-aux (cdr l) seen))))))
```

```
(progn
  (defun ouch ()
    #1=(progn #1#))
  (compile 'ouch))
```

4.6 Reader Security

Extensibility, the ability to make things happen that weren't originally intended or anticipated, is almost always a good thing. In fact, encouraging extensibility wherever possible is what has made lisp as great as it is. However, there are times when we would prefer for things to be as inextensible as possible. In particular, we don't want outsiders to extend themselves into our systems without our knowledge or consent. That is known as being *hacked* or *r00ted*. Today, interesting computing is mostly about communication and networking. When you fully control both programs exchanging data, you obviously trust the entire system. But as soon as there is a possibility for some untrusted party to even partially control one of the programs, the trust system breaks down completely, like a toppling house of cards.

The largest source of these *security* problems arise from what programmers jokingly refer to as *impedance mismatch*. Whenever you use something you don't completely understand, there is a possibility you are using it wrong. There are two approaches to combating impedance mismatches: style (don't use `strcpy(3)`)

and understanding (actually read that manual page). Lisp is a good language for writing secure software because, more so than any other language, lisp always does what is expected. If you just always follow the assumption that lisp does something *right*, you will hardly ever go wrong. For example, if you attempt to write outside the bounds of a string or vector, an obviously problematic situation, lisp will raise an exception and immediately and loudly notify you of the problem. In fact, lisp does this even more *right* than you might expect: after encountering an exception, you have the option of *restarting* your program at another location in your program, preserving most of the state of your computation. In other words, COMMON LISP's exception system doesn't automatically destroy your computation's stack when an exception occurs: you might still want to use that stack. Mostly due to space constraints, the exception system[13] is not described in much detail in this book. Instead, I recommend Peter Seibel's *Practical* COMMON LISP[PRACTICAL-CL].

But part of learning lisp is discovering that everything is extensible. How on earth can we limit this? It turns out that we are thinking about the problem in the wrong direction. As in all areas of computer security, you can't consider defence until you have considered offence. In all other areas of programming, you can arrive at good results constructively, that is by building and using abstractions. In security, you must think destructively. Instead of waiting for and then fixing bugs, you must try to find bugs by breaking your code.

So what attacks are we concerned with? There is no way to attack a program unless you control *input* to that program in some way. Of course in our networked world most programs are pretty useless unless people can give input to them. There are many protocols for shuffling data around the internet[14]. The variety of things we would like to do is simply too vast to create a universal

[13] Actually called the condition system because it is useful for more than just exceptions.

[14] The `nmap-service-probes` file that I help maintain for the Nmap Security Scanner project is one of the most comprehensive and frequently updated data-base of such services.

standard for data interchange. The best that can be done is to provide an extensible framework and allow programmers to customise the protocol to fit the application being created. This will generally mean less network overhead, better transfer algorithms, and more reliability. However, the main advantage is that when we design the protocol we can reduce or eliminate the impedance mismatch which is how to make secure protocols.

The problem with standards for interchanging data is that, in order to support the standard, applications are forbidden from reducing what can be done with the protocol. There is usually some base-line behaviour that must be met in order for an application to conform to the standard. To make secure protocols we need to be able to make sure we accept only what we are certain we can handle and no more.

So what is the lisp way to exchange data? The mechanism for getting data into lisp is called the *lisp reader* and the mechanism for getting it out is called the *lisp printer*. If you have made it this far into the book you already know more than enough to design and use lisp protocols. When you program lisp you are using such a protocol. You interact with lisp by feeding it lisp forms and this often turns out to be the best way to interact with the rest of the world too. Of course you don't trust the rest of the world so precautions must be taken. Remember that to think about security you must think about attacks. The designers of COMMON LISP were thinking about attacks against the reader during design. Earlier in this chapter we described the `#.` read macro that lets the reader execute lisp expressions so we can encode non-serialisable data structures. To mitigate an obvious attack against the lisp reader, COMMON LISP provides `*read-eval*`. From CLtL2:

> Binding `*read-eval*` to `nil` is useful when reading data that came from an untrusted source, such as a network or a user-supplied data file; it prevents the `#.` read macro from being exploited as a "Trojan Horse" to cause arbitrary forms to be evaluated.

When the ANSI COMMON LISP committee voted `*read-eval*` into being in June, 1989, they were thinking like attackers. What

sort of trojan horse would an attacker include? The correct answer is, from a secure software author's point of view, the worst conceivable one you can think of—or worse. Always think that an attacker would like to be able to completely control your system. Traditionally, that means the trojan horse should be something called *shell code*. This is usually a carefully crafted chunk of machine code that does something like provide a unix shell for an attacker to use to further r00t the victim. Crafting this shell code is really an art-form, especially because of the unusual circumstances that such attacks usually exploit. For instance, most shell code cannot contain null bytes because with C-style strings these bytes terminate the string, preventing the inclusion of further shell code. Here is what an example of lisp shell code might look like, assuming the victim is running CMUCL and has Hobbit's original *netcat* (`nc`) [NETCAT] program installed:

```
#.(ext:run-program
    "/bin/nc" '("-e" "/bin/sh" "-l" "-p" "31337"))
```

The above will start listening for connections on the port 31337 and will supply unix shell access to anyone who connects. With traditional exploits, lots of effort is spent on trying to make them as portable and reliable as possible, that is so they will successfully r00t the most amounts of targets the most often. Often this is extremely difficult. In lisp reader attacks, it is extremely easy. Here is how we might update our shell code to make it portable between CMUCL and SBCL:

```
#.(#+cmu ext:run-program
   #+sbcl sb-ext:run-program
    "/bin/nc" '("-e" "/bin/sh" "-l" "-p" "31337"))
```

So the moral is to always make sure you bind `*read-eval*` to `nil` when processing any data that you even slightly distrust. If you rarely use the `#.` read macro, you might be wise to `setq` it to `nil` and only enable it when you expect to use it.

So we can disable the `#.` read macro fairly easily. But is this enough? It depends on your application and what is considered

an effective attack. For interactive programs, this might be sufficient. If we get bad data we will hear about it as soon and loudly as possible. However, for internet servers this is probably not enough. Consider this shell code:

```
)
```

Or this:

```
no-such-package:rewt3d
```

Lisp will normally throw an error because we tried to read in an unbalanced form or lookup a symbol in a package that doesn't exist. Likely our entire application will grind to a halt. This is known as a *denial of service* attack. An even more subtle and more difficult to debug denial of service attack is to pass a circular form using the `##` and `#=` read macros. If our code that processes this data wasn't written with such forms in mind, the result is an impedance mismatch and, likely, a security problem. On the other hand, maybe our application depends on being able to pass circular and shared data structures. The data security requirements depend completely on the application. Luckily, whatever your requirements, the lisp reader and printer are up to the task.

`Safe-read-from-string` is a partial answer to the problem of reader security. This function is less ready for production use than most of the other code in this book. You are advised to think carefully about the security requirements of your application and adapt (or even re-write) this code for your application. `Safe-read-from-string` is a very locked down version of `read-from-string`. It has its own copy of the default lisp *readtable*. This copy has had most of the interesting read macros removed, including the `#` dispatching macro. That means that vectors, bit-vectors, gensyms, circular references, `#.`, and all the rest are out. `Safe-read-from-string` will not even allow keywords or foreign package symbols. It will, however, allow any cons structure, not just well formed lists. It also allows numbers[15] and strings.

[15] Exercise: What is the one class of numbers not allowed?

Listing 4.7: SAFE-READ-FROM-STRING

```
(defvar safe-read-from-string-blacklist
  '(#\# #\: #\|))

(let ((rt (copy-readtable nil)))
  (defun safe-reader-error (stream closech)
    (declare (ignore stream closech))
    (error "safe-read-from-string failure"))

  (dolist (c safe-read-from-string-blacklist)
    (set-macro-character
      c #'safe-reader-error nil rt))

  (defun safe-read-from-string (s &optional fail)
    (if (stringp s)
      (let ((*readtable* rt) *read-eval*)
        (handler-bind
          ((error (lambda (condition)
                    (declare (ignore condition))
                    (return-from
                      safe-read-from-string fail))))
          (read-from-string s)))
      fail)))
```

`Safe-read-from-string` uses lisp's exception system to catch all errors thrown by the lisp `read-from-string` function. If there is any problem with reading from the string, including encountering unbalanced parenthesis or encountering any of the other read macros we have blacklisted in the `safe-read-from-string-blacklist` variable, `safe-read-from-string` will return the value passed as its second argument, or `nil` if none was provided (remember you might want to read in `nil`). Here is how it is typically used[16]:

```
(let* ((g (gensym))
       (v (safe-read-from-string
            user-supplied-string g)))
  (if (eq g v)
    (log-bad-data ; careful how it's logged!
      user-supplied-string)
    (process v)))
```

Of course this version of `safe-read-from-string` is very limited and will probably require modification for your application. In particular, you will probably want keyword symbols. Allowing them is easy: just bind a list without the : character to `safe-read-from-string-blacklist` when you use `safe-read-from-string` and be aware that your symbols might reside in multiple packages (including the `keyword` package). Even if you remove the : character, our above package shell code will be thwarted because we catch all errors during reading, including errors indicating nonexistent packages. `*Read-eval*` is always bound to `nil` in case you decide to remove the # character from the blacklist. If you do so, you might want to create a sub-blacklist for the # dispatching macro (might be a large blacklist). The vertical bar character is blacklisted so that we don't read in wacky looking symbols.

So we can lock down the reader as tightly as we feel necessary, in fact as tight as our application will allow. But even after

[16]Though of course if we were using it in a macro we would use `defmacro!` and its automatic gensyms.

we've made sure that there are no *attack vectors* through the software used to read in a form, how can we minimise the impedance mismatch between what we think the structure of a lisp form is and what it actually can be? We have to verify that it matches up with what we expect. Some data standards call this procedure *validation* against a *schema*, but lisp calls it `destructuring-bind` against an *extended lambda form*. All are terms that try to sound more important than is deserved of the simple concept they represent. The idea is that you want to be sure your data is of the form, or structure, that you expect it to be for some given processing. `Destructuring-bind` checks this structure for us, providing a very useful schema language that includes keywords and optional parameters, and also has the bonus that we get to name the different parts of structure as we go along.

I could give some examples of how to use `destructuring-bind` but it is actually not necessary: we have been using destructuring all along. The argument or parameter list that we insert immediately after the name of a macro when when we use `defmacro`, `defmacro!`, or `destructuring-bind` is called an extended lambda list to highlight the fact that it is more powerful than the destructuring performed for an ordinary lambda list. With extended lambda lists we can *nest* extended lambda lists to destructure list structure of arbitrary depth. Paul Graham's *On Lisp* has an excellent treatment of destructuring. Especially see Graham's `with-places` macro[ON-LISP-P237], preferably after reading *section 6.7, Pandoric Macros.*

So every time you write a macro or a function, you are, in a sense, treating the arguments that this macro or function will receive as data, and the extended or regular lambda list as the schema. In this light, validation of data seems easy. Lisp can validate that our data is structured as it should be and will raise error conditions if not. As above with the reader, when processing data that we even slightly distrust we should think very carefully about the possible attacks and then use lisp's powerful exception and macro systems to construct a validation scheme that allows only the very bare minimum required by the application and maps directly onto how our application works, reducing or eliminating

any impedance mismatch. CL-PPCRE regular expressions are also indispensable for this task. No other language has the potential for secure software that lisp does and this will only become more apparent over time.

Chapter 5

Programs That Program

5.1 Lisp Is Not Functional

One of the most common mischaracterisations of lisp is calling it a functional programming language. Lisp is not functional. In fact it can be argued that lisp is one of the least functional languages ever created. This misunderstanding has interesting roots and is a good example of how a small misnomer can have long-lasting influence and end up causing confusion long after the reasons behind the misnomer have passed into irrelevance. What is a functional language? The only definition that makes sense is

> A functional language is a programming language made up of functions.

So what is a function? A function is a concept from *math* that has been with us for centuries:

> A function is a static, well-defined mapping from input values to output values.

It seems like we use `defun` to define new functions in lisp. For example, the following seems to be a function that uses summation to map the set of all numbers to a new set, one that also includes all numbers:

```
(defun adder (x)
  (+ x 1))
```

Obviously we can apply this object to any number and get the mapped result from its return value, but is `adder` really a function? Well, confusingly, lisp tells us that it is[1]:

```
* (describe #'adder)

#<Interpreted Function> is a function.
```

But calling this object a function is one of the deepest-rooted misnomers in lisp history. What `defun` and lambda forms actually create are *procedures* or, more accurately still, *funcallable instances*[AMOP]. Why the distinction? Procedures don't necessarily have anything to do with mapping values, but instead are pieces of code, possibly with stored environment, that can be executed (funcalled). When lisp programmers write in a certain style called *functional style*, then the resulting procedures can be thought of and combined together pretending that they are math-style functional maps.

The reason why lisp is so often incorrectly described as a functional language has to do with history. Believe it or not but there was once a time when most languages didn't even support the concept of a procedure that modern programmers in any language take for granted. Very early languages didn't provide a suitable abstraction for locally naming arguments in a piece of re-usable code and programmers had to manually allocate registers and manipulate stacks to achieve this behaviour. Lisp, a language that had procedures all along, seemed way more functional than these languages.

Next, once procedural abstraction was given the attention it deserved and incorporated into almost all programming languages, people started to slowly run up against barriers due to

[1] If you haven't already seen the COMMON LISP `describe` function before, try it right now. Describe a function, a special form, a macro, a variable, a symbol, and a closure.

the limited nature of the procedures they had implemented. Programmers started to realise that they would like to be able to return procedures from other procedures, embed them in new environments, aggregate them in data-structures, and, in general, treat them as any old regular value. A slogan sprang up to mobilise programmers for this next big abstraction push: a society without classes, *first-class procedures*. In comparison to languages that relegated the procedure to an inferior second class, lisp, a language that has had these first-class procedures all along, seemed way more functional.

Finally, many languages make a pointless distinction between expression and statement, usually to support some horrible Blub syntax like infix assignment. In lisp, everything returns something[2] and there are no (syntactic) restrictions on what you can nest or combine. It is a simple question with an obvious answer: what is more important in a language, newbie-friendly syntax or real flexibility? Any language that uses infix syntax is reducing the possibilities of its abstractions in many ways. Luckily most modern languages have decided to trust their users enough to allow them to combine expressions mostly as they see fit. In comparison to languages that make these sorts of brain-dead syntax decisions, lisp seems way more functional.

After having become comfortable with this ubiquitous yet misguided terminology, programmers started to realise that the notion of function that had been used in the great functional vs non-functional programming language debate is not only confusing, but is actually fundamentally backwards. To correct this, programmers and academics alike went back to the drawing board, returning to the mathematical definition of a function: a map from input values to output values. If lisp is in any way a functional language, it is only as much so as modern languages such as Perl and Javascript.

Clearly lisp procedures are not functions. Lisp procedures can return non-static values, that is you can call them multiple times with the same arguments and receive different return values each

[2]Except `(values)` which returns nothing. But even this is coerced to `nil` and so can be used in expressions.

time. As in our examples from previous chapters, lisp procedures can store state by closing over variables. Procedures like `rplaca` can change values in memory or elsewhere (such as registers). Lisp procedures like `terpri` and `format` create output (newlines in the case of `terpri`) directed to terminals or files[3]. Lisp procedures like `yes-or-no-p` can read input from a terminal and return values depending on what the user entered. Are these procedures static, well-defined mappings?

Because lisp procedures are not mathematical functions, lisp is not a functional language. In fact, a strong argument can be made that lisp is even less functional than most other languages. In most languages, expressions that look like procedure calls are enforced by the syntax of the language to be procedure calls. In lisp, we have macros. As we've seen, macros can invisibly change the meaning of certain forms from being function calls into arbitrary lisp expressions, a technique which is capable of violating referential transparency in many ways that simply aren't possible in other languages.

After considering that most languages are in fact not at all functional, some language designers decided to find out what programming in a really functional language would be like. As you would expect, programming functional languages is mostly annoying and impractical. Almost no real-world problems can be usefully expressed as static, well-defined mappings from input values to output values. That being said, functional programming is not without merit and many languages have been designed to take advantage of a functional style of programming. What that means is finding a convenient way of isolating the functional parts of a program from the (actually interesting) non-functional parts. Languages like Haskell and Ocaml use this isolation as a means of making aggressive optimisation assumptions.

But this is lisp. We're very non-functional and very proud of it. To the extent that this isolation of side-effects is useful, lisp programmers can and do implement it with macros. The real purpose behind functional programming is to separate the

[3]`Terpri` and `rplaca`—both have reasonable claim to the distinction of worst-named COMMON LISP operator.

functional description of what should happen from the mechanics of how it actually does happen. Lisp is definitely not functional, but, because of macros, there is no better platform or material for implementing functional languages than lisp.

5.2 Top-Down Programming

> You cannot teach beginners top-down programming, because they don't know which end is up.
>
> —C.A.R. Hoare

In *section 3.2, Domain Specific Languages* when we first looked at domain specific languages, we created a simple macro, `unit-of-time`. This macro lets us conveniently specify periods of time in various units with an intuitive, symbol-based syntax:

```
* (unit-of-time 1 d)

86400
```

`Unit-of-time` is a handy domain specific language because the programmer isn't required to remember, for instance, how many seconds are in a day. `Unit-of-time` is implemented with a simple macro that uses a case statement as the heart of the underlying expansion.

An important principle of macro design is *top-down* programming. When designing a lisp macro, you want to start with the abstraction first. You want to write programs that use the macro long before you write the macro itself. Somewhat paradoxically, you need to know how to program in a language before you can write a concise definition/implementation for that language.

So the first step in serious macro construction is always to write *use cases* of the macro, even though there is no way you can test or use them. If the programs written in the new language are comprehensive enough, a good idea of what will be required to implement a compiler or interpreter for the language will follow.

Considering our `unit-of-time` macro, is there a way we can bump this up another level of specification and create a language for creating these sorts of unit convenience macros? Well, `unit-of-time` is a macro, so to do this we will need a macro defining macro...

Stop! That end is up.

We are not starting by considering the implementation of a language, but rather asking ourselves what we would like to use this language for. The answer is that we would like a simple way to define these sorts of unit conversion helpers. In the following example use case, we would like to be able to take a type of unit, `time`, a base unit, seconds, represented here by `s`, and a collection of pairs of a unit and the conversion factor for this unit to the base unit:

```
(defunits% time s
  m 60
  h 3600
  d 86400
 ms 1/1000
 us 1/1000000)
```

`Defunits%` could then expand into code that defines a macro like `unit-of-time` we wrote in *section 3.2, Domain Specific Languages*, allowing us to convert arbitrary time units to seconds. Could it get any better?

This is the point in the design brainstorming that innovation halts in most programming languages. We just created a way to map multiplier values for different units into code that allows us to convert units at our convenience. But a professional lisp programmer recognises that this mapping is a program itself and can be enhanced through the means we regularly enhance lisp programs.

When we are entering many different units it might be helpful to specify units in terms of other units. Let's mandate that the factor used to multiply the unit could also be a list with a value relative to another unit, like so:

```
(defunits%% time s
  m 60
  h (60 m)
  d (24 h)
 ms (1/1000 s)
 us (1/1000 ms))
```

This *chaining* of units just feels natural. Minutes are specified in terms of our base unit, seconds, hours in terms of minutes, and days in terms of hours. To implement this macro in an iterative fashion we first implement unchained behaviour with `defunits%` and next implement chaining with `defunits%%` before add appropriate error checking and settle on our final version, `defunits`.

Notice that this new language can provide more than just a convenient syntax for adding new types of units. This language also allows us to delay the impact of *rounding* in our calculations and also lets lisp use as much exact arithmetic as possible. For instance, the furlong is defined to be exactly 1/8th of a mile, so if we encoded it that way using chaining instead of, say, a metric approximation, we could end up with more accurate results or, perhaps more importantly, results that are as consistent as possible with other calculations using miles. Because we can just add the most precise conversion factor we find and don't have to bother doing any converting ourselves, this macro will let us build conversion routines at a level of expression not possible in other languages.

Using our automatic gensym behaviour described in *section 3.5, Unwanted Capture*, `defunits%` is fairly easy to write. Graham's `symb` function lets us generate the new name for the conversion macro. For example, if the symbol `time` is the type of unit provided then the new conversion macro will be `unit-of-time`. `Defunits%` was constructed by taking our original definition of `unit-of-time`, defined in *section 3.2, Domain Specific Languages*, surrounding it with a `defmacro!` and a backquote, and replacing the parts that need to be regenerated each time the macro is invoked.

Listing 5.1: DEFUNITS-1

```
(defmacro! defunits% (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity) (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
           ((,base-unit) 1)
           ,@(mapcar (lambda (x)
                       `((,(car x)) ,(cadr x)))
                     (group units 2))))))
```

`Defunits%` uses a *nested backquote*—a construct notoriously difficult to understand. Programming with backquotes is almost like having one more dimension of meaning in your code. In other languages, a given programming statement usually has very simple evaluation semantics. You know when every bit of code will run because every bit of code is forced to run at the same time: run-time. But in lisp by nesting backquotes we can scale up and down a *ladder of quotation.* Every backquote we write takes us one step up this ladder: our code is a list that we may or may not evaluate later. But inside the raw list, every comma encountered takes us back down the quotation ladder and actually executes the code from the appropriate step on the ladder[CLTL2-P967].

So there is a simple algorithm for determining when any bit of code in a lisp form will be evaluated. Simply start from the root of the expression and after encountering a backquote, mark up one level of quotation. For every comma encountered, mark down one level of quotation. As Steele notes[CLTL2-P530], following this level of quotation can be challenging. This difficulty, the need to track your current quotation depth, is what makes using backquote feel like another dimension has been added onto regular programming. In other languages you can walk north, south, east, and west, but lisp also gives you the option of going up.

`Defunits%` is a good first step, but still doesn't implement chaining. Right now, the macro implementing this language is mostly a straightforward substitution. Implementing our chaining behaviour will require more sophisticated program logic. Simple substitution won't work because parts of the macro depend on

Listing 5.2: DEFUNITS-CHAINING-1

```
(defun defunits-chaining% (u units)
  (let ((spec (find u units :key #'car)))
    (if (null spec)
      (error "Unknown unit ~a" u)
      (let ((chain (cadr spec)))
        (if (listp chain)
          (* (car chain)
             (defunits-chaining%
               (cadr chain)
               units))
          chain)))))
```

other parts of the macro, so when we build our expansion we need to process the forms provided to the macro completely, not just thinking about it in terms of individual pieces we can splice in.

Remembering that macros are really just functions, we create a utility function for use in the macro definition, `defunits-chaining%`. This utility function takes a unit, as specified by symbols like `S`, `M`, or `H`, and a list of unit specs. We let unit specs either have a single number, which is interpreted as meaning base units, like `(M 60)`, or a list with an indirection to another unit in a chain, like `(H (60 M))`.

This utility function is recursive. In order to find the multiplier against the base unit, we multiply each step in the chain with another call to the utility function to work out the rest of the chain. When the call stack winds back, we get the multiplier used to convert values of a given unit into the base unit. For instance, when we're constructing the multiplier for hours, we find that there are 60 minutes in an hour. We recurse and find there are 60 seconds in a minute. We recurse again and find that seconds is the end of a chain—minutes were specified directly in terms of the base unit. So, winding back from the recursion, we evaluate `(* 60 (* 60 1))`, which is 3600: there are 3600 seconds in an hour.

Listing 5.3: DEFUNITS-2

```
(defmacro! defunits%% (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity) (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
           ((,base-unit) 1)
           ,@(mapcar (lambda (x)
                       `((,(car x))
                         ,(defunits-chaining%
                            (car x)
                            (cons `(,base-unit 1)
                              (group units 2)))))
                     (group units 2))))))
```

With this utility function defined, working out the multiplier for each unit requires only a simple modification to `defunits%`, which we have done in `defunits%%`. Instead of splicing in the value straight from the unit spec, we pass each unit and the entire unit spec to the `defunits-chaining%` utility. As described above, this function recursively works out the multiplier required to turn each unit into the base unit. With this multiplier, `defunits%%` can splice the value into a case statement just like `defunits%`.

As yet, these macros are incomplete. The `defunits%` macro didn't support chaining. `Defunits%%` supported chaining but lacks *error checking*. A professional macro writer always takes care to handle any error conditions that might arise. Cases that involve infinite loops, or are otherwise difficult to debug in the REPL, are especially important.

The problem with `defunits%%` is actually a property of the language we designed: it is possible to write programs that contain cycles. For example:

```
(defunits time s
  m (1/60 h)
  h (60 m))
```

In order to provide proper debugging output, we need to enhance our implementation slightly. Our final version, `defunits`,

Listing 5.4: DEFUNITS

```
(defun defunits-chaining (u units prev)
  (if (member u prev)
    (error "~{ ~a~^ depends on~}"
      (cons u prev)))
  (let ((spec (find u units :key #'car)))
    (if (null spec)
      (error "Unknown unit ~a" u)
      (let ((chain (cadr spec)))
        (if (listp chain)
          (* (car chain)
             (defunits-chaining
               (cadr chain)
               units
               (cons u prev)))
          chain)))))

(defmacro! defunits (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity)
             (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
           ((,base-unit) 1)
           ,@(mapcar (lambda (x)
                       `((,(car x))
                         ,(defunits-chaining
                            (car x)
                            (cons
                              `(,base-unit 1)
                              (group units 2))
                            nil)))
                     (group units 2))))))
```

allows chaining and also provides useful debugging output should a user of the language specify a program with such a cyclic dependency. It can do this because it uses `defunits-chaining`, an improved version of `defunits-chaining%` that maintains a list of all the units that have previously been visited. This way, if we ever visit that unit again through chaining, we can throw an error that concisely describes the problem:

```
* (defunits time s
    m (1/60 h)
    h (60 m))

Error in function DEFUNITS-CHAINING:
  M depends on H depends on M
```

The `defunits` macro is identical to `defunits%%` except that it passes an additional argument `nil` to `defunits-chaining` which is the end of the list representing the history of units that we've already visited. If a new unit is searched for and we've already visited it, a cycle has been detected. We can use this history of units visited to display a helpful message to users of the macro, very probably ourselves, that accidentally write cycles.

So `defunits` is a language specific to the domain of entering units to a conversion routine. Actually it is specific to a much finer domain than that; there are many possible ways it could have been written. Because it is hard to create languages in Blub, and it is easy in lisp, lisp programmers usually don't bother trying to cram everything into one domain. Instead, they just make the language more and more specific to the domain in question until the eventual goal becomes trivial.

An example use of `defunits` is `unit-of-distance`. Just in case you ever wondered, the 1970 adoption of the international nautical mile shortened the fathom, at least to British sailors, by 1/76th:

```
* (/ (unit-of-distance 1 fathom)
     (unit-of-distance 1 old-brit-fathom))
```

Listing 5.5: UNIT-OF-DISTANCE

```
(defunits distance m
  km 1000
  cm 1/100
  mm (1/10 cm)
  nm (1/1000 mm)

  yard 9144/10000 ; Defined in 1956
  foot (1/3 yard)
  inch (1/12 foot)
  mile (1760 yard)
  furlong (1/8 mile)

  fathom (2 yard) ; Defined in 1929
  nautical-mile 1852
  cable (1/10 nautical-mile)

  old-brit-nautical-mile ; Dropped in 1970
    (6080/3 yard)
  old-brit-cable
    (1/10 old-brit-nautical-mile)
  old-brit-fathom
    (1/100 old-brit-cable))
```

```
75/76
```

Which is just over 2 centimetres:

```
* (coerce
    (unit-of-distance 1/76 old-brit-fathom)
    'float)

0.024384
```

5.3 Implicit Contexts

Macros can leverage a technique called *implicit context.* In code that is used frequently, or absolutely must be concise and lacking any surrounding book-keeping cruft, we sometimes choose to implicitly add lisp code around portions of an expression so we don't have to write it every time we make use of an abstraction. We've talked before about implicit contexts and it should be clear that even when not programming macros they are a fundamental part of lisp programming: let and lambda forms have an *implicit progn* because they evaluate, in sequence, the forms in their body and return the last result. `Defun` adds an *implicit lambda* around forms so you don't need to use a lambda form for named functions.

This section describes the derivation and construction of a *code-walking* macro used later in this book, `tree-leaves`[4]. Like `flatten`, this macro inspects a piece of lisp code, considering it a tree, then performs some modifications and returns a new tree. The list structure of the original expression is not modified: `flatten` and `tree-leaves` both cons new structure. The difference between the two is that while the purpose of `flatten` is to remove nested lists and return a flat list that isn't really lisp code, `tree-leaves` preserves the shape of the expression but changes the values of particular atoms.

Let's start off with a simple sketch. `Tree-leaves%` is a function that recurses across the provided `tree` expression, consing a new

[4] Also see the COMMON LISP function `subst`.

Listing 5.6: TREE-LEAVES-1

```
(defun tree-leaves% (tree result)
  (if tree
    (if (listp tree)
      (cons
        (tree-leaves% (car tree)
                      result)
        (tree-leaves% (cdr tree)
                      result))
      result)))
```

list structure with the same shape[5]. When it discovers an atom, instead of returning that atom it returns the value of the `result` argument:

```
* (tree-leaves%
    '(2 (nil t (a . b)))
    'leaf)

(LEAF (NIL LEAF (LEAF . LEAF)))
```

So `tree-leaves%` returns a new tree with all atoms converted into our provided symbol, `leaf`. Notice that the atom `nil` in a car position of a cons cell is not changed, just as it isn't changed when it resides in the cdr position (and represents an empty list).

Changing every element is, of course, pretty useless. What we really want is a way to pick and choose specific atoms and selectively apply transformations to them to get new atoms to insert into the new list structure, leaving atoms we aren't interested in untouched. In lisp, the most straightforward way to write a customisable utility function is to allow *plug-ins* where the user can use custom code to control the utility's behaviour. The `sort` function included with COMMON LISP is an example of this. Here, the less-than function is plugged-in to `sort`:

```
* (sort '(5 1 2 4 3 8 9 6 7) #'<)
```

[5] An empty else clause in an if form returns `nil`, which is also the empty list.

Listing 5.7: PREDICATE-SPLITTER

```
(defun predicate-splitter (orderp splitp)
  (lambda (a b)
    (let ((s (funcall splitp a)))
      (if (eq s (funcall splitp b))
        (funcall orderp a b)
        s))))
```

```
(1 2 3 4 5 6 7 8 9)
```

This concept of taking a function to control behaviour is especially convenient because we can create anonymous functions suited to the task at hand. Or, when we need even more power, we can create functions that create these anonymous functions for us. This is known as *function composition.* While function composition is not nearly as interesting as macro composition[6], it is still an extremely useful technique that all professional lisp programmers must master.

A simple example of function composition is `predicate-splitter`. This function exists to combine two predicates into a single, new predicate. The first predicate takes two arguments and is used for ordering elements. The second predicate takes one argument and determines whether an element is in the special class of elements you want to split your predicate over. For example, in the following we use `predicate-splitter` to create a new predicate that works exactly like less-than except for the fact that even numbers are considered less than odd numbers:

```
* (sort '(5 1 2 4 3 8 9 6 7)
        (predicate-splitter #'< #'evenp))

(2 4 6 8 1 3 5 7 9)
```

[6]Which is why function composition only gets a couple paragraphs where macro composition gets most of this book.

Listing 5.8: TREE-LEAVES-2

```
(defun tree-leaves%% (tree test result)
  (if tree
    (if (listp tree)
      (cons
        (tree-leaves%% (car tree) test result)
        (tree-leaves%% (cdr tree) test result))
      (if (funcall test tree)
        (funcall result tree)
        tree))))
```

So how can we use functions as plug-ins to control how `tree-leaves%` works? In an updated version of `tree-leaves%`, `tree-leaves%%`, we add two different plug-in functions, one to control which leaves to change and one to specify how to convert an old leaf into a new leaf, respectively called `test` and `result`.

We can use `tree-leaves%%` by passing it two lambda expressions, both of which must take a single argument, `x`. In this case we want a new tree: one with identical list structure to our `tree` argument except that all even numbers are changed into the symbol `even-number`:

```
* (tree-leaves%%
    '(1 2 (3 4 (5 6)))
    (lambda (x)
      (and (numberp x) (evenp x)))
    (lambda (x)
      'even-number))

; Note: Variable X defined but never used.

(1 EVEN-NUMBER (3 EVEN-NUMBER (5 EVEN-NUMBER)))
```

It seems to work except lisp correctly calls attention to the fact that we don't make use of the `x` variable in our second plug-in function. When we don't use a variable it is often a sign of a problem in the code. Even when it is deliberate, like here, the

compiler appreciates information on which variables should be ignored. Normally we will make use of this variable but there are cases, like this one, where we actually do not want to. It's too bad we have to take an argument for this function—after all we're just ignoring that argument anyways. This situation comes up often when writing macros designed to be flexible. The solution is to declare to the compiler that it is acceptable to ignore the variable `x`. Because it doesn't hurt to declare a variable ignorable and still use it[7], we may as well declare both `x` variables to be ignorable:

```
* (tree-leaves%%
    '(1 2 (3 4 (5 6)))
    (lambda (x)
      (declare (ignorable x))
      (and (numberp x) (evenp x)))
    (lambda (x)
      (declare (ignorable x))
      'even-number))

(1 EVEN-NUMBER (3 EVEN-NUMBER (5 EVEN-NUMBER)))
```

Here is where this tutorial gets interesting. It seems like `tree-leaves%%` will work fine for us. We can change any leaves in the tree by providing plug-in functions which verify whether the leaf should be changed and what it should be changed to. In a programming language other than lisp, this would be where improvement to the utility stops. But with lisp, we can do better.

Although `tree-leaves%%` provides all the functionality we desire, its interface is inconvenient and redundant. The easier it is to experiment with a utility the more likely we are to find interesting future uses for it. To reduce the surrounding clutter of our code-walking utility, we create a macro that provides implicit context for its users (probably ourselves).

But instead of something simple like an implicit progn or an implicit lambda, we want an entire implicit lexical context that saves us all the overhead of creating these plug-in functions and

[7]Lisp will figure out that it actually can't be ignored.

Listing 5.9: TREE-LEAVES

```
(defmacro tree-leaves (tree test result)
  `(tree-leaves%%
     ,tree
     (lambda (x)
       (declare (ignorable x))
       ,test)
     (lambda (x)
       (declare (ignorable x))
       ,result)))
```

only requires us to enter the bare minimum of code when doing common tasks like translating trees. This implicit lexical context is not like simple implicits in the sense that we didn't just find another use for a common implicit pattern. Instead, we developed a not-so-common pattern, step by step, when we developed our `tree-leaves%%` walking interface.

For the construction of our implicit macro, the `tree-leaves%%` use case in the REPL above was effectively copy-and-pasted directly into the definition of `tree-leaves` and then the parts that we expect to change upon different uses of this macro were parameterised using backquote. Now, through this macro, we have a less redundant interface for using the utility `tree-leaves%%`. This interface is, of course, completely arbitrary since there are many possible ways it could have been written. However, this seems to be the most intuitive, fat-free way, at least for the uses we have so-far envisioned. Macros have allowed us to create an efficient programmer interface in a simple, direct way impossible in other languages. Here is how we can use the macro:

```
* (tree-leaves
    '(1 2 (3 4 (5 . 6)))
    (and (numberp x) (evenp x))
    'even-number)

(1 EVEN-NUMBER (3 EVEN-NUMBER (5 . EVEN-NUMBER)))
```

Notice the variable `x` is actually used without it appearing to

have been defined. That is because there is an *implicit lexical variable* bound around each of the last two expressions. This introduction of a variable without it being visible is said to violate *lexical transparency.* Another way to say it is that an *anaphor* named `x` is introduced for those forms to make use of. We will develop this idea much, much further in *chapter 6, Anaphoric Macros.*

5.4 Code-Walking with Macrolet

> Lisp isn't a language, it's a building material.
>
> —Alan Kay

Forms of expression that are written and seldom spoken, like computer code, often breed diverse pronunciation habits. Most programmers run a dialogue in their heads, reasoning expressions and pronouncing operators, sometimes consciously, often not. For example, the most obvious way to pronounce the name of the lisp special form `macrolet` is simply the audible concatenation of its two lispy component words: macro and let. But after reading Steele's observation[CLTL2-P153] that some lisp programmers pronounce it in a way that rhymes with *Chevrolet*, it can be difficult to get this humorous pronunciation out of your programming dialogue.

However it is pronounced, `macrolet` is a vital part of advanced lisp programming. `Macrolet` is a COMMON LISP special form that introduces new macros into its enclosed lexical scope. Writing `macrolet` syntax transformations is done in the same way as defining global macros with `defmacro`. Just as lisp will expand occurrences of `defmacro`-defined macros in your code, `macrolet`-defined macros will be expanded by the lisp system when it *code-walks* your expressions.

But `macrolet` is not just a convenience. It offers a number of important advantages over using `defmacro` to define macros. First, if you want to have uses of a macro expand differently given their lexical contexts in an expression, creating different `macrolet` contexts is required. `Defmacro` just won't work.

Most importantly, `macrolet` is useful because of how difficult it is to code-walk COMMON LISP expressions. Often we have an arbitrary tree of lisp code, say because we are macro-processing it, and we would like to change the values or meaning of different branches of the tree. To do things like implement temporary meanings for certain forms, and to temporarily over-ride specific macros, perhaps only in particular portions of the lexical context of an expression, we need to walk code. Specifically, we need to recurse through the code, looking for our desired macro or function name in a position where it is being evaluated, and substitute our own expression in its place.

Easy, right? The difficulty is that many legitimate lisp code fragments will break a naive code-walker implementation. Consider if we would like to perform special substitutions for evaluations of a function of a certain symbol, say, `blah`. If we are given the following expression, it is easy to tell that the substitution should occur:

```
(blah t)
```

`Blah` appears in the function position of a list that will get evaluated when the expression is evaluated so we should obviously make the substitution. So far so good. But what if we are passed this form:

```
'(blah t)
```

Because it is quoted, this bit of code is meant to return a literal list. Performing the substitution here would be incorrect. So our code-walkėr must know to stop when it hits a quote and make no substitutions in the form being quoted. Fine, that is easy enough. But let's consider if there are any other situations where it would be incorrect to expand `blah`. What if the code is using `blah` as the name for a lexical variable?

```
(let ((blah t))
  blah)
```

Even though `blah` appears as the first element of a list, here it appears in a local binding for a let form and a use of that binding so should not be expanded. Even this doesn't seem too bad. We could add some special case logic to our code-walker so it knows what to do when it encounters a let form. Unfortunately, we still have 23 more ANSI COMMON LISP special forms[8] that need to have special case logic added for them. What's more, many special forms are complicated to correctly walk. `Let`, as we've already seen, can be tricky, and it gets even worse. The following potentially legal COMMON LISP form contains one use of `blah` that should be expanded. Which one?

```
(let (blah (blah (blah blah)))
  blah)
```

So code-walking is difficult because correctly handling all the special forms is difficult (also see [SPECIAL-FORMS] and [USEFUL-LISP-ALGOS2]). Notice that we don't need any special case logic for forms defined as macros. When a macro is encountered, we can simply expand it until it turns into a function call or a special form. If it is a function, we know it follows lambda's once-only, left-to-right evaluation semantics. It is the special forms that we need to develop special case logic for.

Sounds like a lot of work, doesn't it? It is. A complete COMMON LISP code-walker, especially if designed to be portable, is a large, complicated piece of code. So why doesn't COMMON LISP provide us an interface for code-walking COMMON LISP code? Well, it turns out, in a way, it does, and, in a way, it's called `macrolet`. Code-walking is exactly what your COMMON LISP system needs to do before it can evaluate or compile an expression. Just like our hypothetical code-walker, COMMON LISP needs to understand and handle the special semantics of `let` and other special forms.

Since COMMON LISP has to walk our code in order to evaluate it, there is usually little need for a separate code-walking program. If we want to make selective transformations to expressions in a

[8] There are 25 special forms in ANSI CL, 23 without `let` and `quote`.

way that is smart about what will actually be evaluated, we can simply encode our transformation as a macro and then wrap a `macrolet` form around the expression. COMMON LISP will code-walk this expression when it is evaluated or compiled and will apply the macro transformations specified by the `macrolet`. Of course since `macrolet` defines macros, it does not impose any additional overhead at run-time. Macrolet is for communicating with COMMON LISP's code-walker and the only guarantee that COMMON LISP makes as to when a macro will be expanded is that it will be done before the run-time of a compiled function[CLTL2-P685][ON-LISP-P25].

One of the most common scenarios for using `macrolet` is when you want to pretend that a function is bound in a certain lexical context but you would like the behaviour of using this form to be something other than a function call. `Flet` and `labels` are out—they can only define functions. So our choices are to write a code-walker to look for invocations of this function and replace them with something else, define a global macro with `defmacro` so the "function" will expand into something else, or wrap the form in a `macrolet` and let the system's code-walker sort it out for us.

As mentioned above, writing a code-walker is hard. If at all possible we should avoid that route. Using a global `defmacro` is sometimes possible but often problematic. The largest issue is that COMMON LISP makes few guarantees about when—or how often—macros will be expanded, so we can't reliably use the same name to have different meanings in different lexical contexts. When we overwrite a global macro we can't know if COMMON LISP has expanded—or will want to again expand in the future—old uses of the macro.

For an example of where this code-walking is useful, let's revisit an issue that we swept under the rug in *section 3.3, Control Structures*. Our initial version of the Scheme named let macro, `nlet`, used the `labels` special form to create a new type of control structure. This use of `labels` allowed us to temporarily define a function for use inside the named let body that would allow us to recurse, as if we started the `let` over again with new values for the

Listing 5.10: NLET-TAIL

```
(defmacro! nlet-tail (n letargs &rest body)
  (let ((gs (loop for i in letargs
                  collect (gensym))))
    `(macrolet
       ((,n ,gs
          `(progn
             (psetq
               ,@(apply #'nconc
                        (mapcar
                          #'list
                          ',(mapcar #'car letargs)
                          (list ,@gs))))
             (go ,',g!n))))
       (block ,g!b
         (let ,letargs
           (tagbody
             ,g!n (return-from
                    ,g!b (progn ,@body)))))))))
```

`let` bindings. When we defined this function, we mentioned that because COMMON LISP does not guarantee that it will optimise away tail calls, it is possible that each iteration of this named let control structure will take up needless additional stack space. In other words, unlike in Scheme, COMMON LISP function calls are not guaranteed to be *tail call optimised.*

Even though most decent COMMON LISP compilers will perform proper tail call optimisation, sometimes we need to be certain that the optimisation is being made. The easiest portable way to accomplish this is to change the `nlet` macro so that the expansions it generates necessarily use no unnecessary stack space.

In `nlet-tail`, we surround the supplied body of the macro and wrap it up inside a few forms. We use `block` and `return-from` statements to return the value of the final expression because we are trying to mimic the behaviour of a let form and its implicit progn. Notice that we use a gensym for the name of this block and a gensym for each `let` parameter to avoid unwanted capture

and the loop macro[9] to collect these gensyms.

`Nlet-tail` is used in the same way as our original `nlet`, except that invocations of the named let in non-tail positions are prohibited since they will be expanded to tail calls. Here is the same unimaginative example we used when we presented `nlet`, except that it is guaranteed, even in a lisp that doesn't perform tail call optimisation, to not consume extra stack space:

```
(defun nlet-tail-fact (n)
  (nlet-tail fact ((n n) (acc 1))
    (if (zerop n)
      acc
      (fact (- n 1) (* acc n)))))
```

As this is the motivating example for this section, notice that we use `macrolet` to code-walk the supplied body looking for uses of `fact`. Where our original `nlet` uses the `labels` special form to bind a function, we would like to guarantee that no additional stack space is consumed when we invoke the named let. Technically, we would like to change some bindings in our lexical environment and perform a jump back to the top of the named let. So `nlet-tail` takes the provided name of the let, `fact` in our above usage example, and creates a *local macro* which only takes effect inside the supplied body. This macro expands into code that uses `psetq` to set the bindings of the `let` to the new provided values and then jumps back to the top, no stack space required. And most importantly, we can use the name `fact` for other unrelated macros in our program[10].

To implement this jumping, `nlet-tail` uses a combination of the lisp special forms `tagbody` and `go`. These two forms offer a *goto* system. Although the problems gotos impose upon *structured programming*, whatever that means, are widely discussed, COMMON LISP offers these special forms for exactly the reason we use

[9]`Loop` is oddly one of the most controversial issues in COMMON LISP. Most of the objections to it are, however, completely groundless. `Loop` is a very handy domain specific language for the domain of looping.

[10]What kind of programming book doesn't have a few factorial implementations anyways?

them here. By controlling the *program counter*—the current location in the code we are executing—we can create very efficient macro expansions. While gotos are usually discouraged in modern *high-level* languages, a quick look at any bit of assembly code will show that gotos are very much alive and kicking at the lowest level of our computer software. Even the most adamant anti-goto advocates do not suggest ridding *low-level* languages like C and assembly of goto and jump instructions. It seems that at a low level of programming, we just sort of need gotos, at least to write efficient code.

However, as Alan Kay says, lisp isn't a language—it is a building material. Talking about whether lisp is high or low level just doesn't make sense. There is very high-level lisp like our domain specific languages. With the macros we write to handle these languages, we convert their uses into a lower level of lisp. These expansions are, of course, lisp code too, just not as compressed as our original version. Next, we typically give this medium-level lisp code to a compiler which successively converts it to lower and lower levels of lisp. It won't be long at all before concepts like gotos, conditional branching, and bit fiddling make it into the code, but even then it will still be lisp. Eventually, with a native code compiler, your high level lisp program will have been converted down to assembly language. But even then, chances are that your program will still be lisp. Since most lisp assemblers are written in lisp itself it is only natural to store these assembly programs as lisp objects, resulting in really low-level programs that really are still lisp. It is only once the program is actually assembled into binary machine code that it ceases to be lisp. Or does it?

High and low level distinctions are not applicable to lisp; the level of a lisp program is all a matter of perspective. Lisp is not a language, but instead the most flexible software building material yet discovered.

5.5 Recursive Expansions

When teaching lisp by example to beginners, a question that inevitably arises soon into the lesson is

WTF is a cadr?

There are two ways to proceed at this point. The first is to explain to the student that lisp lists are built of cons cells, each of which has a pointer called car and a pointer called cdr. Once this concept is understood, it is easy to show how the accessor functions for these pointers, also named car and cdr, can be combined into a function called `cadr` that traverses a list and retrieves the second element.

The second approach is to point the student to the COMMON LISP function `second` and ignore `cadr` altogether. Both `cadr` and `second` accomplish the same task: retrieving the second element of a list. The difference is that `second` is named for what it does, and `cadr` is named for how it does it. `Cadr` is *transparently specified.* While `second` is an easy to remember name for the function, it undesirably obscures the meaning of the operation[11]. Transparent specifications are often better because we can think about using the `cadr` function for more than just taking the second element of a list. For instance, we can transparently use `cadr` as a concept for getting the argument destructuring list of a lambda form. `Cadr` and `second` both perform the same task underneath, but can conceptually represent different operations[CLTL2-P530].

Even more important than a philosophical preference for transparent specifications, combinations of `car` and `cdr` can represent more list accessor operations, and more consistently, than the handful of english accessor words. `Car` and `cdr` are useful because we can combine them into new, arbitrary functions. For instance, `(cadadr x)` is the same as `(car (cdr (car (cdr x))))`. COMMON LISP mandates that all combinations of `car` and `cdr` of length four or less must be defined. So although there is no function `second-of-second` for taking the second element of a list and then treating it as a list and retrieving its second element, we can use `cadadr` for this purpose.

It is especially convenient to have these pre-defined combinations of `car` and `cdr` available for functions that take a `:key`

[11] Particularly because `second` is exactly the same as `cadr`: you can't use it to get the second elements of other sequences like vectors.

accessor argument, like `find`:

```
* (find 'a
    '(((a b) (c d)) ((c d) (b a)))
    :key #'cadadr)

((C D) (B A))
```

Using our pre-defined `cadadr` accessor is a more concise than constructing the equivalent lambda expression of a composition of english accessor combinations:

```
* (find 'a
    '(((a b) (c d)) ((c d) (b a)))
    :key (lambda (e)
           (second (second e))))

((C D) (B A))
```

COMMON LISP also provides the functions `nth` and `nthcdr` which can be used as generic accessors if, for instance, we don't know exactly which element we would like to take at compile-time. `Nth` is defined simply: take `n` cdrs of the list and then a car. So `(nth 2 list)`[12] is the same as `(caddr list)` and `(third list)`. `Nthcdr` is identical except that it doesn't do the final car: `(nthcdr 2 list)` is the same as `(cddr list)`.

But if the location in a cons structure is not accessible by one of the above patterns like `nth` or `nthcdr`, we need to combine accessors. Having to combine inconsistent abstractions to accomplish a task is often an indication of incompleteness. Can we define a domain specific language for the domain of accessing lists in order to unite these `car` and `cdr` combining functions, the english accessors, and functions like `nth` and `nthcdr`?

Since `car` and `cdr` are the fundamental operators, our language should involve combining these two accessors in a fully

[12]We can comfortably use `list` as a variable name because of COMMON LISP's second namespace. Such examples would be problematic in single-namespace lisps like Scheme.

Listing 5.11: CXR-1

```
(defmacro cxr% (x tree)
  (if (null x)
    tree
    `(,(cond
         ((eq 'a (cadr x)) 'car)
         ((eq 'd (cadr x)) 'cdr)
         (t (error "Non A/D symbol")))
      ,(if (= 1 (car x))
         `(cxr% ,(cddr x) ,tree)
         `(cxr% ,(cons (- (car x) 1) (cdr x))
                ,tree)))))
```

general way. Because there are an infinite number of such combinations, continuing the combinations by defining functions for every possible accessor is plainly infeasible. What we really want is a single macro that can expand into efficient list traversal code.

The syntax of specifying a list accessor function by naming it starting with a C, followed by one or more A or D characters and ending with an R, is very intuitive and is roughly what we would like to copy for our language. The macro **cxr%** is a pun on these accessors with the one or more A or D characters replaced with an x[13]. With **cxr%**, these As and Ds are specified in a list given as the first argument to the macro. This list is an alternating combination of numbers and either of the symbols A or D.

For example, even though COMMON LISP doesn't provide us an english function to access the eleventh element of a list, we can easily define it:

```
(defun eleventh (x)
  (cxr% (1 a 10 d) x))
```

The point of this section is to illustrate a realistic use for *recursive expansions*. A recursive expansion comes up when a macro expands a form into a new form that also contains a use of

[13]Except in spirit, **cxr** is unrelated to the **cxr** in Maclisp that accessed hunk slots.

the macro in question. Just as with all recursion, this process must terminate on a *base case*. Hopefully the macro will eventually expand into a form that doesn't contain a use of the macro in question and the expansion will finish.

Here we macroexpand an instance of a **cxr%** macro into a form that also uses **cxr%**:

```
* (macroexpand
    '(cxr% (1 a 2 d) some-list))

(CAR (CXR% (2 D) SOME-LIST))
T
```

And when we copy this new recursive form and macroexpand it, we find yet another recursion:

```
* (macroexpand
    '(CXR% (2 D) SOME-LIST))

(CDR (CXR% (1 D) SOME-LIST))
T
```

The results of the next recursion illustrate another possible usage of **cxr%**: the null list accessor[14]:

```
* (macroexpand
    '(CXR% (1 D) SOME-LIST))

(CDR (CXR% NIL SOME-LIST))
T
```

A null list accessor is our base case and expands directly into the accessed list:

```
* (macroexpand
   '(CXR% NIL SOME-LIST))

SOME-LIST
T
```

[14]If COMMON LISP included this, it might be called **cr**.

Using the CMUCL extension `macroexpand-all`, a component of a complete code-walker, we can see the entire expansion of our original `cxr%` form:

```
* (walker:macroexpand-all
    '(cxr% (1 a 2 d) some-list))

(CAR (CDR (CDR SOME-LIST)))
```

Thanks to our excellent lisp compilers, for all intents and purposes this use of `cxr%` is identical to the functions `caddr` and `third`.

But, as the name suggests, `cxr%` is incomplete. It is merely a first sketch of our ultimate macro, `cxr`. The first problem with our sketch is that it only accepts integers as the count of As or Ds. With this specification, there are things that `nth` and `nthcdr` can do that our macro can't.

We need to check for the case where a non-integer is given as the number prefix to an A or D symbol. In this case, our expansion code should evaluate what is provided and use this value[15] as the quantity for the number of cars or cdrs to traverse.

The second problem with `cxr%` is that when given extremely large numbers as the prefix to A or D symbols it will *inline* all the car and cdr combinations. For small numbers, performance can be increased with inlining but generally it doesn't make sense to inline excessively large numbers of cars and cdrs; instead we should use a looping function like `nth` or `nthcdr`.

To fix both of these cases, we add an alternate expansion. If the parameter preceding an A or D symbol isn't an integer, this new behaviour must be used, and if we would rather not inline a large number of cars or cdrs, this behaviour can be selected as well. Arbitrarily choosing this *inline threshold* to be 10, this new behaviour is provided with the macro `cxr`.

With `cxr` we could define `nthcdr` directly in terms of its transparent car and cdr specification:

[15]Hopefully this value should be a number. In lisp, we can safely leave this situation for lisp's exception system to handle and describe to the programmer.

Listing 5.12: CXR

```
(defvar cxr-inline-thresh 10)

(defmacro! cxr (x tree)
  (if (null x)
    tree
    (let ((op (cond
                ((eq 'a (cadr x)) 'car)
                ((eq 'd (cadr x)) 'cdr)
                (t (error "Non A/D symbol")))))
      (if (and (integerp (car x))
               (<= 1 (car x) cxr-inline-thresh))
          (if (= 1 (car x))
            `(,op (cxr ,(cddr x) ,tree))
            `(,op (cxr ,(cons (- (car x) 1) (cdr x))
                       ,tree)))
          `(nlet-tail
             ,g!name ((,g!count ,(car x))
                      (,g!val (cxr ,(cddr x) ,tree)))
             (if (>= 0 ,g!count)
               ,g!val
               ;; Will be a tail:
               (,g!name (- ,g!count 1)
                        (,op ,g!val)))))))))
```

```
(defun nthcdr% (n list)
  (cxr (n d) list))
```

And, similarly, nth:

```
(defun nth% (n list)
  (cxr (1 a n d) list))
```

Because macro writing is an iterative, layer-based process, we are often prompted to *combine* or *compose* macros that we previously implemented. For instance, in the definition of cxr, our alternate expansion makes use of the macro we defined in the previous section: nlet-tail. Nlet-tail is convenient because it allows us to give a name to an iteration construct and, since we only plan on iterating as a tail call, we are certain that we can use it to avoid needless stack consumption.

Here is how the use of cxr in nthcdr% expands:

```
* (macroexpand
   '(cxr (n d) list))

(LET ()
  (NLET-TAIL #:NAME1632
             ((#:COUNT1633 N)
              (#:VAL1634 (CXR NIL LIST)))
    (IF (>= 0 #:COUNT1633)
      #:VAL1634
      (#:NAME1632 (- #:COUNT1633 1)
                  (CDR #:VAL1634)))))
T
```

Notice that complex macro expansions often write code that a human programmer never would. Especially note the use of nil cxrs and the use of a useless let, both left for further macroexpansions and the compiler to optimise away.

Because macros can make more of the expansion visible to the user of the macro, transparent specification is often possible in ways impossible in other languages. For instance, as per the

design of `cxr`, parameters preceding As and Ds that are integers less than `cxr-inline-thresh` will be inlined as calls to `car` and `cdr`:

```
* (macroexpand '(cxr (9 d) list))

(LET ()
  (CDR (CXR (8 D) LIST)))
T
```

But thanks to `cxr`'s transparent specification, we can pass a value that, although not an integer itself, will, when evaluated, become an integer. When we do this, we know that no inlining has taken place because the macro will result in an `nlet-tail` expansion. The simplest form that evaluates to an integer is simply that integer, quoted:

```
* (macroexpand '(cxr ('9 d) list))

(LET ()
  (NLET-TAIL #:NAME1638
             ((#:COUNT1639 '9)
              (#:VAL1640 (CXR NIL LIST)))
    (IF (>= 0 #:COUNT1639)
      #:VAL1640
      (#:NAME1638 (- #:COUNT1639 1)
                  (CDR #:VAL1640)))))
T
```

We often find it useful to combine macros together: `cxr` can expand into a macro we wrote earlier called `nlet-tail`. Similarly, sometimes it is useful to combine a macro with itself, resulting in a recursive expansion.

5.6 Recursive Solutions

It seems the macro we defined in the previous section, `cxr`, has subsumed combinations of the functions `car` and `cdr`, as well

Listing 5.13: DEF-ENGLISH-LIST-ACCESSORS

```
(defmacro def-english-list-accessors (start end)
  (if (not (<= 1 start end))
    (error "Bad start/end range"))
  `(progn
     ,@(loop for i from start to end collect
         `(defun
            ,(symb
               (map 'string
                    (lambda (c)
                      (if (alpha-char-p c)
                        (char-upcase c)
                        #\-))
                    (format nil "~:r" i)))
            (arg)
            (cxr (1 a ,(- i 1) d) arg)))))
```

as the general flat list accessor functions `nth` and `nthcdr`. But what about english accessors like `first`, `second`, and `tenth`? Are these functions useless? Definitely not. When representing the operation of accessing the fourth element in a list, using `fourth` sure beats counting the three Ds in `cadddr`, both for writing and reading efficiency.

In fact, the largest problem with the english accessors is the limitation of having only 10 of them, `first` through `tenth`, in COMMON LISP. But one of the themes of this section, and indeed this book, is that every layer of the lisp onion can use every other layer. In lisp there are no primitives. If we want to define more english accessors, like `eleventh`, we can easily do so, as demonstrated above. The `eleventh` function we defined with `defun` is no different from accessors like `first` and `tenth` specified by ANSI. Since there are no primitives, and we can use all of lisp in our macro definitions, we can take advantage of advanced features like `loop` and `format`[16] in our macro definitions.

The macro `def-english-list-accessors` uses the format

[16] `Format` is a somewhat controversial feature of COMMON LISP. However, like the objections to `loop`, most are based on misunderstanding the concept and scope of domain specific languages.

string "~:r" to convert a number, i, to a string containing the corresponding english word. We change all non-alphabetic characters, as is customary in lisp, to hyphens. We then convert this string to a symbol and use it in a defun form which implements the appropriate accessor functionality with our cxr macro.

For instance, say we suddenly realise we need to access the eleventh element of a list. We can use nth or combinations of cdr and english accessors, but this results in an inconsistent coding style. We could rewrite our code to avoid using english accessors altogether, but there was probably a reason why we chose to use that abstraction in the first place.

Finally, we could define the necessary missing accessors ourself. In other languages, this usually means a lot of copy-pasting or maybe some special-case code generation scripts—neither of which are particularly elegant. But with lisp, we have macros:

```
* (macroexpand
   '(def-english-list-accessors 11 20))

(PROGN
  (DEFUN ELEVENTH (ARG) (CXR (1 A 10 D) ARG))
  (DEFUN TWELFTH (ARG) (CXR (1 A 11 D) ARG))
  (DEFUN THIRTEENTH (ARG) (CXR (1 A 12 D) ARG))
  (DEFUN FOURTEENTH (ARG) (CXR (1 A 13 D) ARG))
  (DEFUN FIFTEENTH (ARG) (CXR (1 A 14 D) ARG))
  (DEFUN SIXTEENTH (ARG) (CXR (1 A 15 D) ARG))
  (DEFUN SEVENTEENTH (ARG) (CXR (1 A 16 D) ARG))
  (DEFUN EIGHTEENTH (ARG) (CXR (1 A 17 D) ARG))
  (DEFUN NINETEENTH (ARG) (CXR (1 A 18 D) ARG))
  (DEFUN TWENTIETH (ARG) (CXR (1 A 19 D) ARG)))
T
```

Being able to create these english accessors reduces the impact of the ten accessor limitation of ANSI COMMON LISP. If we ever need more english accessors, we just create them with the def-english-list-accessors macro.

How about ANSI's limitation of only defining combinations of car and cdr up to a depth of four? Sometimes, when program-

Listing 5.14: CXR-CALCULATOR

```
(defun cxr-calculator (n)
  (loop for i from 1 to n
        sum (expt 2 i)))
```

ming complicated list processing programs, we wish there was an accessor function defined that isn't. For instance, if we are using the function `cadadr`, `second-of-second`, to access a list and we change our data representation so that the references now need to be `second-of-third`, or `cadaddr`, we encounter this COMMON LISP limitation.

Like we did with the english accessors, we could write a program that defines extra combinations of `car` and `cdr`. The problem is that, unlike english accessors, an increase in the depth of a combination function like `caddr` results in an exponential increase in the number of functions that need to be defined. Specifically, the number of accessors that need to be defined to cover a depth of `n` can be found by using the function `cxr-calculator`.

We see that ANSI specifies 30 combinations:

```
* (cxr-calculator 4)

30
```

To give you an idea of how quickly the number of functions required grows:

```
* (loop for i from 1 to 16
        collect (cxr-calculator i))

(2 6 14 30 62 126 254 510 1022 2046
 4094 8190 16382 32766 65534 131070)
```

Obviously to cover all the combinations of `car` and `cdr` in deep `cxr` functions we need an approach different from how we tackled the english accessor problem. Defining all combinations of `car` and `cdr` up to some acceptable depth is infeasible.

Listing 5.15: CXR-SYMBOL-P

```
(defun cxr-symbol-p (s)
  (if (symbolp s)
    (let ((chars (coerce
                   (symbol-name s)
                   'list)))
      (and
        (< 6 (length chars))
        (char= #\C (car chars))
        (char= #\R (car (last chars)))
        (null (remove-if
                (lambda (c)
                  (or (char= c #\A)
                      (char= c #\D)))
                (cdr (butlast chars)))))))))
```

To start with, we should have a solid specification of what a cxr symbol is. Cxr-symbol-p is a concise definition: all symbols that start with C, end with R, and contain 5 or more As or Ds in-between. We don't want to consider cxr symbols with less than 5 As or Ds because those functions are already guaranteed to be defined by COMMON LISP[17].

Next, because we plan on using cxr to implement the functionality of arbitrary car and cdr combinations, we create a function cxr-symbol-to-cxr-list to convert a cxr symbol, as defined by cxr-symbol-p, into a list that can be used as the first argument to cxr[18]. Here is an example of its use:

```
* (cxr-symbol-to-cxr-list
    'caddadr)

(1 A 1 D 1 D 1 A 1 D)
```

Notice the use of the function list* in cxr-symbol-to-cxr-list. List* is almost the same as list except that its last ar-

[17]Re-binding functions specified by COMMON LISP is forbidden.

[18]Amusingly, the deprecated function explode might prove useful in this situation but was left out of COMMON LISP because nobody could think of good uses for it.

Listing 5.16: CXR-SYMBOL-TO-CXR-LIST

```
(defun cxr-symbol-to-cxr-list (s)
  (labels ((collect (l)
             (if l
               (list*
                 1
                 (if (char= (car l) #\A)
                   'A
                   'D)
                 (collect (cdr l))))))
    (collect
      (cdr       ; chop off C
        (butlast ; chop off R
          (coerce
            (symbol-name s)
            'list))))))
```

gument is inserted into the cdr position of the last cons cell in the created list. `List*` is very convenient when writing recursive functions that build up a list where each stack frame might want to add more than one element to the list. In our case, each frame wants to add two elements to the list: the number 1 and one of the symbols `A` or `D`.

Finally, we decide the only way to effectively provide cxr functions of an arbitrary depth is to code-walk provided expressions and define only the necessary functions. The `with-all-cxrs` macro uses Graham's `flatten` utility to code-walk the provided expressions in the same way that the `defmacro/g!` macro from *section 3.5, Unwanted Capture* does. `With-all-cxrs` finds all the symbols satisfying `cxr-symbol-p`, creates the functions they refer to using the `cxr` macro, and then binds these functions around the provided code with a `labels` form[19].

Now we can enclose expressions within the forms passed to `with-all-cxrs` and pretend that these expressions have access to any possible cxr function. We can, if we choose, simply return these functions for use elsewhere:

[19]The one problem with this approach is that such accessors will not be `setf`able.

Listing 5.17: WITH-ALL-CXRS

```
(defmacro with-all-cxrs (&rest forms)
  `(labels
     (,@(mapcar
          (lambda (s)
            `(,s (l)
               (cxr ,(cxr-symbol-to-cxr-list s)
                    l)))
          (remove-duplicates
            (remove-if-not
              #'cxr-symbol-p
              (flatten forms)))))
     ,@forms))
```

```
* (with-all-cxrs #'cadadadadadr)

#<Interpreted Function>
```

Or, as shown in the following macro expansion, we can embed arbitrarily complex lisp code that makes use of this infinite class:

```
* (macroexpand
   '(with-all-cxrs
      (cons
        (cadadadr list)
        (caaaaaaaar list))))

(LABELS
  ((CADADADR (L)
     (CXR (1 A 1 D 1 A 1 D 1 A 1 D) L))
   (CAAAAAAAAR (L)
     (CXR (1 A 1 A 1 A 1 A 1 A 1 A 1 A 1 A) L)))
  (CONS
    (CADADADR LIST)
    (CAAAAAAAAR LIST)))
T
```

Often a task that sounds difficult—like defining the infinite classes of english list accessors and car-cdr combinations—is re-

ally just a collection of simpler problems grouped together. In contrast to single problems that happen to be difficult, collections of simpler problems can be tackled by approaching the problem recursively. By thinking of ways to convert a problem into a collection of simpler problems, we employ a tried-and-true approach to problem-solving: *divide and conquer.*

5.7 Dlambda

In our discussion of closures we alluded to how a closure can be used as an object, and how, generally, indefinite extent and lexical scope can replace complicated object systems. But one feature that objects frequently provide that we have mostly ignored up until now is multiple *methods.* In other words, while our simple counter closure example only allows one operation, increment, objects usually want to be able to respond to different *messages* with different behaviours.

Although a closure can be thought of as an object with exactly one method—`apply`—that one method can be designed so as to have different behaviours based on the arguments passed to it. For instance, if we designate the first argument to be a symbol representing the message being passed, we can provide multiple behaviours with a simple `case` statement based on the first argument.

To implement a counter with an increment method and a decrement method, we might use this:

```
(let ((count 0))
  (lambda (msg)
    (case msg
      ((:inc)
        (incf count))
      ((:dec)
        (decf count)))))
```

Notice that we have chosen *keyword symbols*, that is symbols that begin with a : and always evaluate to themselves, to indicate messages. Keywords are convenient because we don't have

Listing 5.18: DLAMBDA

```
(defmacro! dlambda (&rest ds)
  `(lambda (&rest ,g!args)
     (case (car ,g!args)
       ,@(mapcar
           (lambda (d)
             `(,(if (eq t (car d))
                    t
                    (list (car d)))
               (apply (lambda ,@(cdr d))
                      ,(if (eq t (car d))
                           g!args
                           `(cdr ,g!args)))))
           ds))))
```

to quote them or export them from packages, and are also intuitive because they are designed to perform this and other sorts of *destructuring*. Often in a lambda or defmacro form keywords are not destructured at run-time. But since we are implementing a message passing system, which is a type of *run-time destructuring*, we leave the keyword processing operation to be performed at run-time. As previously discussed, destructuring on symbols is an efficient operation: a mere pointer comparison. When our counter example is compiled it might be reduced down to the following machine code:

```
2FC:    MOV    EAX, [#x582701E4]    ; :INC
302:    CMP    [EBP-12], EAX
305:    JEQ    L3
307:    MOV    EAX, [#x582701E8]    ; :DEC
30D:    CMP    [EBP-12], EAX
310:    JEQ    L2
```

But to make this convenient, we would like to avoid having to write the case statement for every object or class we create. Situations like this deserve macros. The macro I like to use is `dlambda`, which expands into a lambda form. This expansion includes a way for one of many different branches of code to be

executed depending on the arguments it is applied to. This sort of run-time destructuring is what gives `dlambda` its name: it is either a *destructuring* or a *dispatching* version of `lambda`.

`Dlambda` is designed to be passed a keyword symbol as the first argument. Depending on which keyword symbol was used, `dlambda` will execute a corresponding piece of code. For instance, our favourite example of a closure—the simple counter—can be extended so as to either increment or decrement the count based on the first argument using `dlambda`. This is known as the *let over dlambda* pattern:

```
* (setf (symbol-function 'count-test)
    (let ((count 0))
      (dlambda
        (:inc () (incf count))
        (:dec () (decf count)))))

#<Interpreted Function>
```

We can increment

```
* (count-test :inc)

1
```

and decrement

```
* (count-test :dec)

0
```

the closure depending on the first argument passed. Although left empty in the above let over dlambda, the lists following the keyword symbols are actually *lambda destructuring* lists. Each dispatch case, in other words each keyword argument, can have its own distinct lambda destructuring list, as in the following enhancement to the counter closure:

```
* (setf (symbol-function 'count-test)
    (let ((count 0))
      (dlambda
        (:reset () (setf count 0))
        (:inc (n) (incf count n))
        (:dec (n) (decf count n))
        (:bound (lo hi)
          (setf count
            (min hi
                 (max lo
                      count)))))))

#<Interpreted Function>
```

We now have several different possible lambda destructuring lists that might be used, depending on our first keyword argument. `:reset` requires no arguments and brings **count** back to 0:

```
* (count-test :reset)

0
```

`:inc` and `:dec` both take a numeric argument, **n**:

```
* (count-test :inc 100)

100
```

And `:bound` ensures that the value of **count** is between two *boundary values*, `lo` and `hi`. If **count** falls outside this boundary it is changed to the closest boundary value:

```
* (count-test :bound -10 10)

10
```

An important property of `dlambda` is that it uses lambda for all destructuring so as to preserve the normal error checking and

debugging support provided by our COMMON LISP environment. For instance, if we give only one argument to `count-test` we will get an error directly comparable to an incorrect *arity* lambda application:

```
* (count-test :bound -10)

ERROR: Wrong argument count, wanted 2 and got 1.
```

Especially when `dlambda` is embedded into a lexical environment forming a closure, `dlambda` allows us to program—in object oriented jargon—as though we are creating an object with multiple *methods*. `Dlambda` is tailored so as to make this functionality easily accessible while not departing from the syntax and usage of lambda. `Dlambda` still expands into a single lambda form and, as such, its evaluation results in exactly what evaluating `lambda` results in: an *anonymous function* that can be saved, applied, and, most importantly, used as the lambda component of a lexical closure.

But `dlambda` takes this synchronisation with lambda one step further. In order for `dlambda` to provide as smooth a transition from code containing the `lambda` macro as possible, `dlambda` also allows us to process invocations of the anonymous function that don't pass a keyword argument as the first symbol. When we have large amounts of code written using the closure through a normal lambda interface we would appreciate being able to add special case dlambda methods without changing how the rest of the code uses the interface.

If the last possible method is given the symbol `t` instead of a keyword argument, the provided method will always be invoked if none of the special case keyword argument methods are found to apply. Here is a contrived example:

```
* (setf (symbol-function 'dlambda-test)
    (dlambda
      (:something-special ()
        (format t "SPECIAL~%"))
      (t (&rest args)
```

```
        (format t "DEFAULT: ~a~%" args))))

#<Interpreted Function>
```

With this definition, the majority of ways to call this function invoke the default case. Our default case uses the `&rest` lambda destructuring argument to accept all possible arguments; we are free to narrow the accepted arguments by providing more specific lambda destructuring parameters.

```
* (dlambda-test 1 2 3)
DEFAULT: (1 2 3)
NIL
* (dlambda-test)
DEFAULT: NIL
NIL
```

However, even though this anonymous function acts mostly like a regular lambda form defined with the default case, we can pass a keyword argument to invoke the special method.

```
* (dlambda-test :something-special)
SPECIAL
NIL
```

A key feature, one that will be exploited heavily by the following chapter, is that both the default method and all the special methods are, of course, invoked in the lexical context of the encompassing `dlambda`. Because of how closely `dlambda` is integrated with lambda notation, this lets us bring multi-method techniques to the domain of creating and extending lexical closures.

Chapter 6

Anaphoric Macros

6.1 More Phors?

Some of the most interesting macros from Paul Graham's *On Lisp* are *anaphoric macros.* An anaphoric macro is one that deliberately captures a variable from forms supplied to the macro. Thanks to their *transparent specifications*, these deliberately captured variables allow us windows of control over the macro expansion. Through these windows we can manipulate the expansion through *combinations.*

The classic anaphora like those in *On Lisp* are named after the literal words[1] *anaphor* and its plural, *anaphora.* An anaphor is a means of capturing a *free U-language word* for use in subsequent U-language. In programming terms, implementing a classic anaphor means finding places in your code—or in code you would like to write—where expressions can benefit from being able to refer to the results of previous, related expressions. Graham's anaphora and associated code are worth careful study. Especially see the **defanaph** macro[ON-LISP-P223] which enables some interesting types of *automatic anaphor* programming.

After some period of use, **alambda** has been found to be the most useful of the anaphoric macros in *On Lisp.* It is also one of the most simple and elegant demonstrations of an anaphoric

[1] A U-language quotation.

Listing 6.1: ALAMBDA

```
;; Graham's alambda
(defmacro alambda (parms &body body)
  `(labels ((self ,parms ,@body))
     #'self))
```

macro and its intentional variable capture.

With `alambda` we capture the name `self` so that we can use it to refer to the very anonymous function we are building. In other words, recursing is as simple as a call to `self`. For example, the following function returns a list[2] of the numbers from `n` down to 1:

```
(alambda (n)
  (if (> n 0)
    (cons
      n
      (self (- n 1)))))
```

`Alambda` lets our code be intuitive and easy to read, and allows us to change our mind about whether an anonymous function should be able to call itself as easily as adding a single letter[3]. Because of `alambda`'s transparent specification for the `self` binding—and the fact that the only reason to ever use `alambda` is to make use of this binding—unwanted variable capture is never a problem.

Another handy anaphoric macro from *On Lisp* is `aif`, a macro that binds the result of the test clause to `it` for the true (secondary or consequent) clause to make use of[4]. `Aif` makes use of a valuable COMMON LISP feature: *generalised booleans*. In COMMON

[2]If the condition is false, an absent tertiary clause on an if form returns `nil`, which is a list.

[3]This is another reason to not sharp-quote lambda forms. Changing a sharp quoted lambda form to an alambda form also requires deleting two characters.

[4]Exercise: Why would the false (tertiary or alternant) clause never make use of this anaphor?

Listing 6.2: AIF

```
;; Graham's aif
(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))
```

LISP, any non-nil value is a true boolean value so COMMON LISP programmers typically embed interesting information in truth values. Languages that have reserved true and false values—notably Scheme—employ *explicit booleans*, which sometimes force you to throw out extra information to satisfy redundant type constraints. Scheme has actually added a *kludge* to allow `if`, `cond`, `and`, `or`, and `do` to accept non-boolean[5] values[R5RS-P25]. COMMON LISP, of course, is designed *right*—everything is a boolean.

It must also be pointed out that `aif` and `alambda`, like all anaphoric macros, violate *lexical transparency*. A fashionable way of saying this is currently to say that they are *unhygienic* macros. That is, like a good number of macros in this book, they invisibly introduce lexical bindings and thus cannot be created with macro systems that strictly enforce hygiene. Even the vast majority of Scheme systems, the platform that has experimented the most with hygiene, provide unhygienic defmacro-style macros—presumably because not even Scheme implementors take hygiene very seriously. Like training wheels on a bicycle, hygiene systems are for the most part toys that should be discarded after even a modest level of skill has been acquired.

Yes, there are many interesting things we can do with deliberate variable capture. There are a lot more phors. This book and Graham's *On Lisp* describe a only a tiny fraction of the potential inherent to this technique. Many more incredible inventions will come out of the intelligent application of anaphoric macros.

After a brief interlude into anaphora introduced by read macros, the remainder of this chapter describes a modest, specific application of anaphora related to one of the central themes of

[5] According to the Scheme `boolean?` predicate.

Listing 6.3: SHARP-BACKQUOTE

```
(defun |#`-reader| (stream sub-char numarg)
  (declare (ignore sub-char))
  (unless numarg (setq numarg 1))
  `(lambda ,(loop for i from 1 to numarg
                  collect (symb 'a i))
     ,(funcall
        (get-macro-character #\`) stream nil)))

(set-dispatch-macro-character
  #\# #\` #'|#`-reader|)
```

this book: the lexical closure—*let over lambda.* Most of this chapter describes interesting anaphoric macros for customising, adapting, and extending closures. Although the topics are very practical for use in real code, their main purpose is as a platform for discussing the properties and variations of anaphoric macros. Using macros to extend the concept of a closure is currently a hot research topic[FIRST-CLASS-EXTENTS][OPENING-CLOSURES].

6.2 Sharp-Backquote

Although most anaphora are introduced by regular macros, read macros also have the potential to introduce code that invisibly creates bindings for us. When read macros do so, they are called *read anaphora.* This section presents one such read macro that, while itself very modest, surprised even myself by turning out to be one of the most consistently useful throughout this book. I have tried to introduce this macro as soon as possible so that it can be used for the remainder of the code. Already, several macros shown should have used it.

Sharp-backquote is a read macro that reads in as a lambda form. By default, this lambda form will take exactly one argument: `a1`. The read macro then recursively invokes the `read` function with the supplied stream. Here is an example with the evaluation stopped (by `quote`) so we can observe the transparent

introduction of the read anaphor[6]:

```
* '#`((,a1))

(LAMBDA (A1)
  `((,A1)))
```

This read macro abstracts out a common macro pattern. For example, if we have a list of variables and would like to make a list of let bindings that bind each variable to a symbol, say, empty, we can use mapcar like so:

```
* (mapcar (lambda (a)
            (list a ''empty))
    '(var-a var-b var-c))

((VAR-A 'EMPTY)
 (VAR-B 'EMPTY)
 (VAR-C 'EMPTY))
```

But especially for complicated list structure, this can get messy, so lisp programmers like to use backquote to knock it up one level of quotation:

```
* (mapcar (lambda (a)
            `(,a 'empty))
    '(var-a var-b var-c))

((VAR-A 'EMPTY)
 (VAR-B 'EMPTY)
 (VAR-C 'EMPTY))
```

Our new anaphor-introducing read macro hides the lambda form:

```
* (mapcar #`(,a1 'empty)
    '(var-a var-b var-c))
```

[6]The prefix of the captured symbol, "a", of course stands for anaphor.

```
((VAR-A 'EMPTY)
 (VAR-B 'EMPTY)
 (VAR-C 'EMPTY))
```

The reason for the 1 character in the symbol a1 above is that users of the read macro can introduce a variable number of anaphora depending on the number provided to the numarg parameter of the read macro:

```
* '#2`(,a1 ,a2)

(LAMBDA (A1 A2)
  `(,A1 ,A2))
```

So we can mapcar sharp-backquote expressions across more than one list at a time:

```
* (let ((vars '(var-a var-b var-c)))
    (mapcar #2`(,a1 ',a2)
      vars
      (loop for v in vars
            collect (gensym
                      (symbol-name v)))))

((VAR-A '#:VAR-A1731)
 (VAR-B '#:VAR-B1732)
 (VAR-C '#:VAR-C1733))
```

Another way to think about sharp-backquote is that it is to list interpolation as the format function is to string interpolation. Just as format lets us use a template with slots that are to be filled with the values of separate arguments, sharp-backquote lets us separate the structure of the list interpolation from the values we want to splice in. Because of the earlier described *duality of syntax* between lambda forms in the function position of a list and the lambda forms that use the lambda macro to expand into a function, we can also use sharp-backquote as the first element in a function call:

```
* (#3`(((,a1)) ,@a2 (,a3))
     (gensym)
     '(a b c)
     'hello)

(((#:G1734)) A B C (HELLO))
```

Unlike `format`, sharp-backquote doesn't use sequential positioning. Instead it uses the number on our anaphoric bindings. As a consequence, the order can be mixed up and we can even splice in bindings more than once:

```
* (#3`(((,@a2)) ,a3 (,a1 ,a1))
     (gensym)
     '(a b c)
     'hello)

(((A B C)) HELLO (#:G1735 #:G1735))
```

Exercise: The references to the gensym `#:G1735` look like they point to the same symbol but, of course, you never really can tell with gensyms by looking at their print names. Are these symbols eq? Why or why not?

6.3 Alet and Finite State Machines

With *lambda* and *if* there is only one useful anaphoric configuration. But the most interesting types of anaphoric macros make use of expansions in unforeseen ways. This section—even most of this chapter—is based around one such macro: `alet`. What extra bindings could be useful to forms inside the body of a let form? The very purpose of `let` is to create such bindings so capturing the variable introductions given to a let form is already done. However, a macro enhancement to `let` can have complete access to all the forms given to it, even the body of expressions intended to be evaluated with the new bindings. So what is the most useful part of the body? In most cases it is the last form in the body since

Listing 6.4: ALET-1

```
(defmacro alet% (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     this))
```

the results from that form will be returned from the let statement itself[7]. We've seen that when we return a lambda expression that references these bindings created by `let`, the result is a lexical closure—an object frequently stored and used to later access the variables in the let statement. So, extending our closure-object analogy, the `alet%` macro acts exactly like the `let` special form except that it captures the symbol `this` from the body and binds it to the last expression in the form's body—the one that will be returned as the closure[8].

`Alet%` can be useful when we have initialisation code in the lambda form that we don't want to duplicate. Because `this` is bound to the lambda form that we are returning, we can execute it before we return it from the enclosing let. The following is a closure whose construction shows a simple example use of `alet%` that avoids duplicating its reset and initialisation code:

```
* (alet% ((sum) (mul) (expt))
    (funcall this :reset)
    (dlambda
      (:reset ()
        (psetq sum 0
               mul 1
               expt 2))
      (t (n)
        (psetq sum (+ sum n)
               mul (* mul n)
```

[7]Because `let` provides an implicit progn.

[8]`Setq` is used so that the form bound to `this` is defined in the lexical scope of the other arguments given by `letargs`.

```
                expt (expt expt n))
        (list sum mul expt))))

#<Interpreted Function>
```

Which we can call successively to change the values of **sum**, **mul**, and **expt**:

```
* (loop for i from 1 to 5 collect (funcall * 2))

((2 2 4)
 (4 4 16)
 (6 8 256)
 (8 16 65536)
 (10 32 4294967296))
```

We can now reset the closure by invoking its **:reset** method. Notice that we only had to write the reset base cases (0 for **sum**, 1 for **mul**, and 2 for **expt**) in one location thanks to **alet%**:

```
* (funcall ** :reset)

NIL
```

Now that the closure's variables are reset, we can see a new sequence from the start:

```
* (loop for i from 1 to 5 collect (funcall *** 0.5))

((0.5 0.5 1.4142135)
 (1.0 0.25 1.1892071)
 (1.5 0.125 1.0905077)
 (2.0 0.0625 1.0442737)
 (2.5 0.03125 1.0218971))
```

Note that **alet%** changes the evaluation order of the forms in the let body. If you look at the expansion you will notice that the last form in the body is actually evaluated first, and its results

Listing 6.5: ALET

```
(defmacro alet (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (lambda (&rest params)
       (apply this params))))
```

are then bound to the lexical binding `this` before the preceding forms are evaluated. As long as the last argument is a constant, however, this re-ordering makes no difference. Remember that a lambda expression[9] is a constant value and is thus perfectly suitable for use in `alet%`.

As with many macro enhancements, because of the many degrees of freedom available, improvements to this macro are counterintuitive. Although there are many possibilities, this section considers one such specific improvement. `Alet%` can be made to not return the last form in its body—which we anticipate to be a lambda form—but instead a function that looks up another function inside the let form's lexical scope, then calls that function instead. This is sometimes called *indirection* because instead of returning a function to do something, we return a function that looks up a function using a pointer dereference, then uses that function instead. Indirection is a concept ubiquitous throughout programming languages for good reason. It lets us change things at run-time that, without indirection, are fixed at compile-time. Lisp lets us use indirection in a more succinct and efficient manner than many other programming languages. `Alet`, a version of `alet%` with indirection added, allows the function we returned as the closure to now be accessed or replaced by code inside the alet body, or, if we use `dlambda`, as will be shown soon, even outside the alet body.

Now that we can change the function that gets executed when invoking a closure with our `alet` macro we can create a pair of mutually referential functions using a pattern called *alet over*

[9] `Dlambda` expands into lambda forms.

alambda. As long as all the states go back to the original state—instead of going to each other—alet over alambda is a convenient way to specify nameless state machines.

The following is a typical counter closure that takes an argument **n** and can have its direction toggled between increment and decrement by **n** when we pass the symbol **invert** as the argument instead of a number:

```
* (alet ((acc 0))
    (alambda (n)
      (if (eq n 'invert)
        (setq this
              (lambda (n)
                (if (eq n 'invert)
                  (setq this #'self)
                  (decf acc n))))
        (incf acc n))))

#<Interpreted Function>
```

Let's store this closure so we can use it as often as we want:

```
* (setf (symbol-function 'alet-test) *)

#<Interpreted Function>
```

When we start, we are going up:

```
* (alet-test 10)

10
```

But we can change the actual function to be called to the internal lambda expression in our definition by passing the symbol **invert** to the closure:

```
* (alet-test 'invert)

#<Interpreted Function>
```

And now we're going down:

```
* (alet-test 3)

7
```

Finally, thanks to the `self` binding provided by `alambda`, we can again change the function to be called by passing the symbol `invert`:

```
* (alet-test 'invert)

#<Interpreted Function>
```

Back where we started, going up:

```
* (alet-test 5)

12
```

This closure has been bound in the function namespace to the symbol `alet-test`. But this closure is slightly different than a regular closure. While both this closure and regular closures are pointers to a single environment, one that can have any number of references to it, this closure uses indirection to change which piece of code gets run when it is invoked. Although any piece of code can be installed, only ones in the lexical scope of the `alet`, the one with the `this` anaphor available, can access its lexical bindings. But there is still nothing to prevent us from installing a new closure, with its own lexical bindings and perhaps with changed behaviour in the *indirection environment* installed by `alet`. Much of the remainder of this chapter is about useful things we can do with indirection environments created by `alet`.

A common macro technique is informally known as *turning a macro inside out*. When you turn a macro inside out you pick a typical form that uses a macro similar to the macro you would like to create, and expand it. You then use that expansion as a template for your desired macro. For example, we would like

a more general method of creating closures with multiple states than the alet over alambda counter presented earlier. Here is the above inside out expansion of the invertible counter alambda use case:

```
* (macroexpand
   '(alambda (n)
      (if (eq n 'invert)
        (setq this
              (lambda (n)
                (if (eq n 'invert)
                  (setq this #'self)
                  (decf acc n))))
        (incf acc n))))

(LABELS ((SELF (N)
           (IF (EQ N 'INVERT)
             (SETQ THIS
                   (LAMBDA (N)
                     (IF (EQ N 'INVERT)
                       (SETQ THIS #'SELF)
                       (DECF ACC N))))
             (INCF ACC N))))
  #'SELF)
```

If we re-factor the above expansion slightly to take advantage of the fact that `labels` allows us to create multiple function bindings[10], we arrive at the following:

```
(alet ((acc 0))
  (labels ((going-up (n)
             (if (eq n 'invert)
               (setq this #'going-down)
               (incf acc n)))
           (going-down (n)
             (if (eq n 'invert)
```

[10]Hence the plurality of `labels`.

Listing 6.6: ALET-FSM

```
(defmacro alet-fsm (&rest states)
  `(macrolet ((state (s)
                `(setq this #',s)))
     (labels (,@states) #',(caar states))))
```

```
                (setq this #'going-up)
                (incf acc (- n)))))
    #'going-up))
```

From this, we notice that `alambda` can use the `labels` special form to make all of its bindings available to all of the bodies of its functions. What's more, we now have a fairly complete template for our eventual macro.

`Alet-fsm` gives us a convenient syntax for expressing multiple possible *states* for our closure to exist in. It is a very light sugar coating of macros over top of `labels`, combined with a *code-walking* `macrolet` transformation that allows us to pretend as though we have a function, `state`, that can change the closure's current state, accessed through the `this` anaphor provided by `alet`. As an example, here is a cleaner version of our invertible counter:

```
(alet ((acc 0))
  (alet-fsm
    (going-up (n)
      (if (eq n 'invert)
        (state going-down)
        (incf acc n)))
    (going-down (n)
      (if (eq n 'invert)
        (state going-up)
        (decf acc n)))))
```

`Alet-fsm` is an instance of a technique we haven't seen before: *anaphor injection.* The use of this anaphor violates lexical

transparency in so many ways that it is actually, somehow, *lexically invisible.* Not only does `alet` bind `this` for us invisibly, but the use of `this` by our `alet-fsm` macro is similarly invisible. `Alet-fsm` injects a free variable into our lexical context without us being able to see it at all in the lexical context.

The stylistic issues of this are uncertain[11], but macro programming is, of course, not about style. It is about power. Sometimes free variable injection can create a symbiosis between two macros—one that can better programmatically construct expansions than can two isolated expansions. Because this type of macro programming is sophisticated, parallels can again be drawn to the C pointer analogy. Just as learning C pointers breeds dubious stylistic advice, so does free variable injection.

The most plausible hypothesis for the source of difficulty in understanding free variable injection is its *fail-safe* behaviour[12]. With an anaphor, if the supplied user code doesn't make use of the binding the code will probably continue to function, whether you intended it to or not. It has, possibly, failed silently and thus un-safely. However, when you inject a free variable and there is no environment there to capture it, your entire expression has become free. When this happens, you need to decide what to do before you can evaluate that expression. It has failed safe.

Style aside, free variable injection is sometimes just what we need when we want two related macros to communicate back and forth. Injection is really the same operation as that performed by anaphora, just in the opposite direction. Because you are opening up a new channel of communication between your macros, the complexity issues scale even more quickly. Consider sitting in a house full of fragile glass. You can safely throw objects to people outside the house, even if they don't bother catching them, but you had better make sure you catch any objects thrown back at you.

[11] As are, by nature, all stylistic issues. Once something is perfectly understood, style becomes irrelevant. Free variable injection is not yet perfectly understood.

[12] Safe in the sense that, contrary to the real world, failing as quickly and as loudly as possible is safest.

Listing 6.7: ICHAIN-BEFORE

```
(defmacro! ichain-before (&rest body)
  `(let ((,g!indir-env this))
     (setq this
       (lambda (&rest ,g!temp-args)
         ,@body
         (apply ,g!indir-env
                ,g!temp-args)))))
```

6.4 Indirection Chains

There are many ways we can take advantage of the **this** anaphor provided by **alet**. Since the environment is accessed through a dummy closure that forwards all invocations to the real closure pointed to by **this**, we can pass the dummy closure reference around, copying it as often as needed. *Indirection* like this is useful because we can change what happens when this dummy closure is invoked without having to change references to the dummy closure.

Ichain-before is intended to be expanded in an **alet** form. It adds a new body of code to be executed before invoking the main closure. Going back to our counter example, **ichain-before** lets us add a new closure that prints out the previous value of the closed-over **acc** variable before it goes ahead and increments it:

```
* (alet ((acc 0))
    (ichain-before
      (format t "Changing from ~a~%" acc))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
```

Which works as expected:

```
* (funcall * 2)
Changing from 0
```

```
2
* (funcall ** 2)
Changing from 2
4
```

There is a reason we put chain in the name of `ichain-before`, though. We can put as many of these closures on to be executed as we please:

```
* (alet ((acc 0))
    (ichain-before
      (format t "A~%"))
    (ichain-before
      (format t "B~%"))
    (ichain-before
      (format t "C~%"))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
```

Each addition of a new link in the chain adds the link to the very beginning of the chain, resulting in the links being visited in the reverse order from which they were added:

```
* (funcall * 2)
C
B
A
2
```

Statically adding indirection chains is sometimes useful when changing macros to avoid re-structuring macros by adding new surrounding code. But the most interesting possibilities for indirection chains pop up when we add them dynamically. Because we can create new closures at run-time and because we can access the internals of a closure through an anaphor, we can re-write how

functions work at run-time. Here is a simple example in which every invocation of the closure adds another bit of code that prints "Hello world" when run:

```
* (alet ((acc 0))
    (lambda (n)
      (ichain-before
        (format t "Hello world~%"))
      (incf acc n)))

#<Interpreted Function>
```

Every invocation adds a new closure to the indirection chain:

```
* (loop for i from 1 to 4
    do
      (format t "~:r invocation:~%" i)
      (funcall * i))
first invocation:
second invocation:
Hello world
third invocation:
Hello world
Hello world
fourth invocation:
Hello world
Hello world
Hello world
```

The `ichain-after` macro is similar to the `ichain-before` macro except it adds the closures to the other end of the execution chain: after the main closure has been invoked. `Ichain-after` uses `prog1`, which executes its provided forms consecutively and then returns the result of evaluating the first form.

`Ichain-before` and `ichain-after` can be combined so that the before forms are executed before the evaluation of the main closure and the after forms after:

Listing 6.8: ICHAIN-AFTER

```
(defmacro! ichain-after (&rest body)
  `(let ((,g!indir-env this))
     (setq this
       (lambda (&rest ,g!temp-args)
         (prog1
           (apply ,g!indir-env
                  ,g!temp-args)
           ,@body)))))
```

```
* (alet ((acc 0))
    (ichain-before
      (format t "Changing from ~a~%" acc))
    (ichain-after
      (format t "Changed to ~a~%" acc))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
* (funcall * 7)
Changing from 0
Changed to 7
7
```

Ichain-before and ichain-after are macros that inject free variables into their expansion. They inject the symbol this which we rely on being captured by the expansion of an alet macro. This sort of injection of symbols might seem to be bad style or error-prone, but it is actually a common macro technique. In fact, almost all macros inject symbols into the expansion. For instance, along with this, the macro ichain-before also injects symbols like let, setq, and lambda to be spliced into wherever the macro is expanded. The difference between symbols like this and predefined symbols like setq is that while lambda always refers to a single well-understood ANSI macro, symbols like this can refer to different things depending on the environments in which they are expanded.

Listing 6.9: ICHAIN-INTERCEPT-1

```
(defmacro! ichain-intercept% (&rest body)
  `(let ((,g!indir-env this))
     (setq this
       (lambda (&rest ,g!temp-args)
         (block intercept
           (prog1
             (apply ,g!indir-env
                    ,g!temp-args)
             ,@body))))))
```

`ichain-before` and `ichain-after` are useful for tagging on code for a closure to run before or after the execution of the original closed-over expression but are by no means the only things we can do with the `this` anaphor. Another common task is checking for the validity of closure data after an invocation of the closure.

`ichain-intercept%` is another macro designed to be used inside an `alet` form. The idea is that we would like to be able to intercept invocations of the closure and verify that the actions they performed didn't cause some sort of inconsistent state in the closure.

So we can add an intercept to our usual counter closure like so:

```
* (alet ((acc 0))
    (ichain-intercept%
      (when (< acc 0)
        (format t "Acc went negative~%")
        (setq acc 0)
        (return-from intercept acc)))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
```

When the counter falls below 0, the code installed by `ichain-intercept%` will warn us:

Listing 6.10: ICHAIN-INTERCEPT

```
(defmacro! ichain-intercept (&rest body)
  `(let ((,g!indir-env this))
     (setq this
       (lambda (&rest ,g!temp-args)
         (block ,g!intercept
           (macrolet ((intercept (v)
                        `(return-from
                           ,',g!intercept
                           ,v)))
             (prog1
               (apply ,g!indir-env
                      ,g!temp-args)
               ,@body)))))))
```

```
* (funcall * -8)
Acc went negative
0
```

The counter was reset back to 0:

```
* (funcall ** 3)

3
```

The most interesting thing about `ichain-intercept%` is that it introduces a *block anaphor* named `intercept`. To use this anaphor we use `return-from`. The block will return this value from the closure invocation, intercepting the original value.

Instead of capturing the block anaphor `intercept`, `ichain-intercept` creates a local macro that allows the code inside `ichain-intercept` to use `intercept` to expand into a `return-from` where the block is specified by a gensym.

```
* (alet ((acc 0))
    (ichain-intercept
      (when (< acc 0)
        (format t "Acc went negative~%")
```

```
        (setq acc 0)
        (intercept acc)))
    (lambda (n)
      (incf acc n)))

#<Interpreted Function>
```

This works the same as with `ichain-intercept%`:

```
* (funcall * -8)
Acc went negative
0
* (funcall ** 3)

3
```

Of course, introducing all these closures transparently into operations can affect run-time performance. Luckily, modern lisp compilers are very good at optimising closures. If your application can stand a few pointer dereferences—and often it can—indirection chains might just be the best way to structure it. See *section 7.4, Pointer Scope* for another interesting way to think about indirection chains. Also see CLOS's before, after, and around functionalities.

6.5 Hotpatching Closures

There are three purposes for this important section. First, another interesting use of the `this` anaphor from `alet` is described. Second, the pattern *alet over dlambda* is discussed. Finally, a useful macro technique called *anaphor closing* is introduced.

In order to clearly illustrate anaphor closing, we will not work with the `alet` macro but instead an inside out expansion. `Alet-hotpatch%` is an expansion of `alet` with a special lambda form provided. This lambda form checks the first argument[13] to see

[13]With a pointer comparison.

Listing 6.11: ALET-HOTPATCH-1

```
(defmacro alet-hotpatch% (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (lambda (&rest args)
       (if (eq (car args) ':hotpatch)
         (setq this (cadr args))
         (apply this args)))))
```

if it is the keyword symbol **:hotpatch** and, if so, replaces the indirected closure with another provided argument.

Being able to change the closure used in another forwarding closure at run-time is known as *hotpatching*. For instance, here we create a hotpatchable closure and store it in the symbol-function cell of the symbol **hotpatch-test** for later use:

```
* (setf (symbol-function 'hotpatch-test)
    (alet-hotpatch% ((acc 0))
      (lambda (n)
        (incf acc n))))

#<Interpreted Function>
```

It can now be used like so:

```
* (hotpatch-test 3)

3
* (hotpatch-test 4)

7
```

We can replace the lambda form—along with its associated environment—by calling this closure with the symbol **:hotpatch** and a replacement function or closure:

```
* (hotpatch-test
```

Listing 6.12: ALET-HOTPATCH

```
(defmacro alet-hotpatch (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (dlambda
       (:hotpatch (closure)
         (setq this closure))
       (t (&rest args)
         (apply this args)))))
```

```
    :hotpatch
    (let ((acc 0))
      (lambda (n)
        (incf acc (* 2 n)))))

#<Interpreted Function>
```

Now the closure will have the new, hotpatched behaviour:

```
* (hotpatch-test 2)

4
* (hotpatch-test 5)

14
```

Notice how the counter value reset to 0 since we also hotpatched the closure's environment with a new value for the counter's accumulator, `acc`.

Haven't we seen this sort of *run-time destructuring* on keyword symbols before? Yes, in fact we wrote a macro for doing exactly this in *section 5.7, Dlambda.* `Alet-hotpatch` is a version of `alet-hotpatch%` that takes advantage of `dlambda`. Sometimes without even realising it, by using macros we wrote previously in the definition of new macros, we are performing *macro combination.* With well designed macros the expansion can be fully understood and, although it might violate lexical transparency in

Listing 6.13: LET-HOTPATCH

```
(defmacro! let-hotpatch (letargs &rest body)
  `(let ((,g!this) ,@letargs)
     (setq ,g!this ,@(last body))
     ,@(butlast body)
     (dlambda
       (:hotpatch (closure)
         (setq ,g!this closure))
       (t (&rest args)
         (apply ,g!this args)))))
```

many ways, no combination problems emerge because all components fit together meaningfully.

Alet-hotpatch creates a hotpatchable closure but there is one slight conceptual flaw. Because the only real reason for using **alet-hotpatch** is to create this sort of hotpatchable closure, we might forget that it also introduces the anaphor **this** into the scope of the forms provided. When we forget about anaphora we've created, we risk unwanted variable capture problems. To avoid these problems, we might choose to employ a technique known as *anaphor closing*. When we close an anaphor, we don't need to change the way our anaphoric macros function, just restrict them in the ways they can be combined.

Because we have turned the **alet** expansion inside out, we can lexically see the creation of the **this** anaphor in the definition of **alet-hotpatch**. And because **alet-hotpatch** also contains the code using the **this** anaphor to implement hotpatching, we can close the anaphor so that the symbol **this** is no longer captured by the macro. How do we normally avoid introducing unwanted bindings? We name the bindings using gensyms of course.

Let-hotpatch is an example of closing the **this** anaphor into a more contained version—a safer version for when hotpatching is all that is required. The leading a was removed from the name to suggest that this new macro no longer introduces an anaphor into the supplied body of code. Of course if we wanted to refer to **this** for some reason other than hotpatching, we should have left the anaphor open.

This technique of opening and closing anaphora becomes second nature after you have written enough of such macros. Just like we can write macros that inject free variables into an expansion without thinking about how we will capture them until we write the lexical context in which they will be expanded, we sometimes choose to leave an anaphor open while developing macros to experiment with combinations of anaphoric macros and free variable injection macros. Once the most useful combinations are found, we can merge the macros together and replace all anaphora used during development with with gensyms. Like `let-hotpatch` does, this technique can use `defmacro!` to move the anaphor's scope from the macro expansion to the macro definition. Instead of lexically introducing an anaphor, we introduced another type of anaphor—one that doesn't take effect in the full lexical scope of the expansion but only in another, more limited scope. This scope is described further in the following section.

6.6 Sub-Lexical Scope

Our `defmacro!` macro-defining macros that we defined in *section 3.5, Unwanted Capture* look for the presence of automatic gensyms in the provided code using Graham's `flatten` utility. Now is the time to confess a small lie we have been telling through this book. Before now, because we hadn't explained free variable injection and anaphora, we pretended that the G-bang symbol names in `defmacro!` definitions are applicable in the lexical scope of the macro definition. This is actually not true—`defmacro!` provides these bindings under a slightly different type of scope called *sub-lexical scope.*

Remember that scope means where references to a variable are valid and lexical scope means that the name is applicable to code in the textual body of a binding construct such as `let`. The important distinction between lexical scope and sub-lexical scope is that for lexical scope this includes all macroexpansions of code in the `let` body. So describing lexical scope as creating variables only accessible to code in the textual body of a binding construct is actually a lie too—macros can *inject* variable references. Such

variables are injected from outside the textual body of the binding construct.

Implementing genuine textual scoping by limiting the possible ways to access lexical variables results in sub-lexical scope. References to a sub-lexically scoped variable are only valid if the symbols representing them occur in the raw lists that were passed to lisp before macro-expansion.

Because `defmacro!` pre-processes the code it was given and creates the list of all the G-bang symbols before the code gets expanded, the G-bang symbols are sub-lexically bound. We can't write macros that inject G-bang symbols into `defmacro!` because lexical bindings for the G-bang symbols were never created. Here is a typical use of a sub-lexical G-bang symbol:

```
* (defmacro! junk ()
    `(let ((,g!var))
       ,g!var))

JUNK
```

Both G-bang symbols were found in the sub-lexical scope of `defmacro!` so expansion is as we would expect:

```
* (macroexpand '(junk))

(LET ()
  (LET ((#:VAR1663))
    #:VAR1663))
T
```

However, in order to explore the concept of sub-lexical scope, we will define a macro that injects a G-bang symbol:

```
* (defmacro injector-for-g!var ()
    ''g!var)

INJECTOR-FOR-G!VAR
```

Now we can write **junk2**. **Junk2** is identical to **junk** except that we have replaced our G-bang symbols with a macro that expands into a G-bang symbol:

```
* (defmacro! junk2 ()
    `(let ((,(injector-for-g!var)))
       ,(injector-for-g!var)))

JUNK2
```

But since the G-bang symbols are sub-lexically bound—and thus don't look into the macro expansions of forms—**defmacro!** doesn't convert the symbols into automatic gensyms:

```
* (macroexpand '(junk2))

(LET ()
  (LET ((G!VAR))
    G!VAR))
T
```

Although the above code will still function, sub-lexically scoped variable references can break expressions when some references that refer to a variable exist in sub-lexical scope and others don't:

```
* (defmacro! junk3 ()
   `(let ((,g!var))
      ,(injector-for-g!var)))

JUNK3
* (macroexpand '(junk3))

(LET ()
  (LET ((#:VAR1672))
    G!VAR))
T
```

Sub-lexical scoping turns up surprisingly often in complex macros. As well as `defmacro!`, we've seen it in at least one other example: the `with-all-cxrs` macro from *section 5.6, Recursive Solutions* sub-lexically binds list accessor functions. The consequence of sub-lexical binding is that we can't refer to such bindings from macro expansions. Sometimes this access limitation is useful, sometimes not. In `with-all-cxrs`, sub-lexicality could be considered undesirable. When our accessor is in `with-all-cxrs`'s sub-lexical scope, there is no problem:

```
* (with-all-cxrs
    (cadadadr nil))

NIL
```

And we can even write macros that expand into these accessors, as long as the macro definitions are in the sub-lexical scope of `with-all-cxrs`:

```
* (with-all-cxrs
    (macrolet ((accessor (l)
                 `(cadadadr ,l)))
      (accessor nil)))

NIL
```

But notice that `with-all-cxrs` binds the accessor function sub-lexically so we can't define a macro to inject the accessor:

```
* (macrolet ((accessor (l)
                `(cadadadr ,l)))
    (with-all-cxrs
      (accessor nil)))

This function is undefined: CADADADR
```

Now that we are familiar with *anaphora* and have seen numerous examples of complex macros—including some that utilise

Listing 6.14: LET-BINDING-TRANSFORM

```
(defun let-binding-transform (bs)
  (if bs
    (cons
      (cond ((symbolp (car bs))
             (list (car bs)))
            ((consp (car bs))
             (car bs))
            (t
             (error "Bad let bindings")))
      (let-binding-transform (cdr bs)))))
```

sub-lexical scope—we can discuss an interesting theoretical macro: `sublet`. This macro is designed to create sub-lexical bindings for code using a syntax similar to the usual let form syntax. The discussion of `sublet`, as with many lisp macros, begins with a utility.

`Let-binding-transform` is a simple utility that handles the case of a let form binding being a single symbol. In the following, `a` is normalised to `(a)`:

```
* (let-binding-transform
    '(a (b) (c nil)))

((A) (B) (C NIL))
```

`Sublet` also uses the `tree-leaves` utility we defined in *section 5.3, Implicit Contexts.* Recall that the `tree-leaves` macro takes three arguments: an arbitrary list structure, an expression that can make use of an `x` variable to determine whether a leaf should be changed, and another expression that can make use of a different `x` to determine what valid leaves should be changed to.

The choice to implicitise the bindings of `x` with the same name turns out to be a useful *duality of syntax.* When we can't factor common code in an expression the usual way, sometimes we can gain this brevity advantage by using syntactic duals in other ways. The definition of `sublet` uses the self-referential read macros described in *section 4.5, Cyclic Expressions.* Especially for things

Listing 6.15: SUBLET

```
(defmacro sublet (bindings% &rest body)
  (let ((bindings (let-binding-transform
                    bindings%)))
    (setq bindings
      (mapcar
        (lambda (x)
          (cons (gensym (symbol-name (car x))) x))
        bindings))
    `(let (,@(mapcar #'list
                     (mapcar #'car bindings)
                     (mapcar #'caddr bindings)))
       ,@(tree-leaves
           body
           #1=(member x bindings :key #'cadr)
           (caar #1#)))))
```

like accessors that can change many times throughout the writing of a program, read macros allow us to have one and only one form representing the accessor. Thanks to our use of *implicitisation* with the `tree-leaves` macro, it is easy to find and understand the code duplication because the code is close together.

`Sublet` takes the form representing the let bindings and applies our `let-binding-transform` utility, generating new list structure in the process. It then prepends[14] a gensym to each binding with a print name corresponding to the binding name. `Sublet` expands into a let form which binds these gensym symbols to the values passed into the binding form, then uses `tree-leaves` to replace all occurrences of the binding name symbols in the provided code with their corresponding gensyms. `Sublet` does not expand any macros or parse any special forms in the body to look for occurrences of these binding name symbols because `sublet` creates sub-lexical bindings. For example, if all references to `a` are sub-lexical, it will replace them with gensyms:

```
* (macroexpand
```

[14]Prepends instead of appends so we can still support bindings without default values, such as `(a)`.

```
   '(sublet ((a 0))
      (list a)))

(LET ((#:A1657 0))
  (LIST #:A1657))
T
```

However, because sub-lexical scope doesn't involve expanding macros, and thus necessarily doesn't involve interpreting special forms like quote, instances of the symbol a that aren't supposed to be variable references are also changed:

```
* (macroexpand
   '(sublet ((a 0))
      (list 'a)))

(LET ((#:A1658 0))
  (LIST '#:A1658))
T
```

Sub-lexical scope takes effect before list structure is interpreted as lisp code by your system's code-walker. That is an important observation—one with ramifications still not completely explored. `Sublet` interprets your code differently than does the code-walker provided with COMMON LISP.

Here we are standing on one of the many edges of macro understanding. What sort of interesting types of scoping lie between unexpanded sub-lexical scope and fully expanded lexical scope? For lack of a better name, we will call this infinitely large category of scopes *super sub-lexical scopes*[15].

A fairly obvious super sub-lexical scope uses `sublet*`. This macro uses `sublet` underneath but changes each form in the body by macro expanding them with the `macroexpand-1` function. Now, instead of appearing in the raw list structure, references to symbols must occur after the first step of macro expansion. This

[15] I am giving it this silly name because I expect better names to become obvious when the concept is better understood.

Listing 6.16: SUBLET*

```
(defmacro sublet* (bindings &rest body)
  `(sublet ,bindings
     ,@(mapcar #'macroexpand-1 body)))
```

type of super sub-lexical scope allows a macro in each of the let form's body to inject or remove references from the scope. If the macros don't do either of these things—or if the forms aren't macros at all—this type of super sub-lexical scope acts just like sub-lexical scope:

```
* (macroexpand
   '(sublet* ((a 0))
      (list a)))

(LET ((#:A1659 0))
  (LIST #:A1659))
T
```

But we can define another injector macro to test this super sub-lexical scope:

```
* (defmacro injector-for-a ()
    'a)

INJECTOR-FOR-A
```

Sublet* will expand this injector macro:

```
* (macroexpand-1
   '(sublet* ((a 0))
      (injector-for-a)))

(SUBLET ((A 0))
  A)
T
```

Which will then be interpreted sub-lexically by `sublet`, meaning that the injected variable a exists within the type of super sub-lexical scope provided by `sublet*`:

```
* (macroexpand-1 *)

(LET ((#:A1663 0))
  #:A1663)
```

But nested macros in the expression are not expanded by `macroexpand-1` so `sublet*` doesn't put them into sub-lexical scope for `sublet` to see:

```
* (macroexpand-1
   '(sublet* ((a 0))
      (list (injector-for-a))))

(SUBLET ((A 0))
  (LIST (INJECTOR-FOR-A)))
T
```

So a is not captured sub-lexically[16]:

```
* (walker:macroexpand-all *)

(LET ((#:A1666 0))
  (LIST A))
```

With `sublet` and `sublet*` we can control at what level of macro expansion the a variable is considered valid by using sub-lexical or super sub-lexical scopes. As mentioned above, super sub-lexical scope is actually an infinite class of scopes, one that is almost completely unexplored intellectually. As many ways as there are to walk code (a lot) there are super sub-lexical scopes. This class of scoping leads into another category of mostly unexplored macros: macros that change how lisp macros work, when

[16] `Walker:macroexpand-all` is a CMUCL component of a complete code-walker.

they are expanded, where references are valid, how special forms are interpreted, etc. Eventually, a macro-programmable macro expander.

6.7 Pandoric Macros

Pandora's box is a Greek myth about the world's first woman: Pandora. Pandora, the U-language symbol, translates from Greek into all-gifted. Pandora, the woman, was tempted by curiosity to open a small box which irreparably unleashed all of humanity's evil and sins upon the world. While the macros described in this section are very powerful and might teach you a way of programming you never forget, rest assured that our outcome will be far better than poor Pandora's. Open the box.

First we take a slight detour through another famous lisp book: *Lisp in Small Pieces*[SMALL-PIECES] by Christian Queinnec. Queinnec is a widely respected lisp expert and has contributed much to our lisp knowledge. Queinnec's book is about implementing compilers and interpreters of varying sophistication in and for the Scheme programming language[17].

In *Lisp in Small Pieces* there is a short but interesting discussion on macros. Much of it relates to describing the different macro system variations possible thanks to the ambiguity of the Scheme macro specification[18], but there are also a few interesting notes on why we might want to use macros and how to go about using them. If you have read and understood *chapter 3, Macro Basics*, most of the macros presented in the chapter of *Lisp in Small Pieces* about macros will, to you, belong in the trivial category, save one enticing macro that we will now discuss.

Like many programming books, *Lisp in Small Pieces* takes us to and leaves us at an implementation of a system for *object-oriented* programming. Usually these implementations serve to outline a subset of *CLOS*, the COMMON LISP Object System. Queinnec calls his subset *MEROONET*. Queinnec remarks that

[17]Though it sometimes describes other lisps and their features.

[18]Thanks, but no thanks.

when defining a method for a MEROONET class it would be nice to be able to directly refer to the fields of the object being defined instead of using accessors. In Queinnec's (translated) words[SMALL-PIECES-P340-341]:

> Let's take the case, for example, of the macro `with-slots` from CLOS; we'll adapt it to a MEROONET context. The fields of an object—let's say the fields of an instance of `Point`—are handled by read and write functions like `Point-x` or `set-Point-y!`. It would be simpler to handle them directly by the name of their fields, `x` or `y`, for example, in the context of defining a method.

Here is Queinnec's desired interface (which he has called `define-handy-method`) defining a new method, `double`:

```
(define-handy-method (double (o Point))
  (set! x (* 2 x))
  (set! y (* 2 y))
  o )
```

Which is more pleasing for programmers than the otherwise necessary MEROONET syntax:

```
(define-method (double (o Point))
  (set-Point-x! o (* 2 (Point-x o)))
  (set-Point-y! o (* 2 (Point-y o)))
  o )
```

In other words, it would be nice if we could use macros to access foreign bindings—in this case object slots—as if they were lexical bindings. Although this is undeniably useful for abbreviation purposes, its most important implication is its ability to give *dualities of syntax* to our existing and future macros.

As Queinnec notes, COMMON LISP implements this functionality for CLOS with a macro called `with-slots`. This is an example of COMMON LISP doing what it was designed to do: allowing abstractions based on a refined, standardised macro system. While most languages are designed to be easy to implement,

Listing 6.17: PANDORICLET

```
(defmacro pandoriclet (letargs &rest body)
  (let ((letargs (cons
                   '(this)
                   (let-binding-transform
                     letargs))))
    `(let (,@letargs)
       (setq this ,@(last body))
       ,@(butlast body)
       (dlambda
         (:pandoric-get (sym)
           ,(pandoriclet-get letargs))
         (:pandoric-set (sym val)
           ,(pandoriclet-set letargs))
         (t (&rest args)
           (apply this args))))))
```

COMMON LISP is designed to be powerful to program. Queinnec's conclusion was that language limitations make this mostly impossible in Scheme, especially where portability is required:

> [L]acking reflective information about the language and its implementations, we cannot write a portable code-walker in Scheme, so we have to give up writing **define-handy-method**.

Although COMMON LISP still allows a large number of legal ways to implement macro systems, it is designed to provide general meta-programming tools that come together in standard and portable ways. The two advanced COMMON LISP macro features that allow us to implement things like CLOS's **with-slots** are *generalised variables* and *symbol macros*. This section takes advantage of an opportunity to show off this wonderful confluence of COMMON LISP features as well as to bring together everything we have seen so far regarding anaphoric macros, in the process discovering an interesting class of macros called *pandoric macros*.

The idea behind **pandoriclet** is to *open closures*, allowing their otherwise closed-over lexical variables to be accessed externally. As with some of our previous macros like **alet-hotpatch**,

`pandoriclet` compiles an indirection environment that chooses different run-time behaviours depending on the arguments passed.

We again started with an *inside out* expansion of `alet`, keeping in mind the introduction of an anaphor called `this`. `Pandoriclet` is similar to other macros we've seen. As with all of our anaphoric `let` variants, we assume the final form in the `pandoriclet` body will be a lambda form. Like `alet-hotpatch`, `pandoriclet` uses the `dlambda` macro to dispatch between different possible pieces of code to execute when the closure returned from `pandoriclet` is invoked. `Pandoriclet` uses the `let-binding-transform` utility function introduced in the previous section to deal with null bindings created—like `(let (a) ...)`. This utility function is necessary to `pandoriclet` for the same reason that it was necessary for `sublet`: these macros code-walk the bindings provided to `let` where our previous macros blindly spliced the bindings into another `let`.

We have put in two calls to list-creating utility functions yet to be defined: `pandoriclet-get` and `pandoriclet-set`, each of which accept a list of let bindings. Notice we can reference functions that don't yet exist as long as we define them before the macro is expanded which obviously can't occur before we use the macro. Using auxiliary functions to help with defining macros is a good habit to pick up. Not only can it make your definitions more readable, but also can help when testing components of the macro and can prove useful in future macros. The best part about this sort of abstraction is that, as when combining macros, we keep our lexical context available for utilities to make use of.

So, remembering the lexical context, we write `pandoriclet-get` and `pandoriclet-set`. For `pandoriclet-get`, we remember that `dlambda` has bound a variable `sym` around where our list will be spliced in. We use `sym` in a `case` form that compares it to the symbols that were passed to `pandoriclet`[19]. If we find a symbol, the current value of the binding it refers to is returned. If not, an error is thrown. `Pandoriclet-set` is nearly identical, except that `dlambda` bound one extra symbol for it to use: `val`.

[19]Recall that `case` with symbols compiles to a single pointer comparison per case.

Listing 6.18: PANDORICLET-ACCESSORS

```
(defun pandoriclet-get (letargs)
  `(case sym
     ,@(mapcar #`((,(car a1)) ,(car a1))
               letargs)
     (t (error
          "Unknown pandoric get: ~a"
          sym))))

(defun pandoriclet-set (letargs)
  `(case sym
     ,@(mapcar #`((,(car a1))
                  (setq ,(car a1) val))
               letargs)
     (t (error
          "Unknown pandoric set: ~a"
          sym val))))
```

`Pandoriclet-set` uses `setq` to change the binding referred to by `sym` to `val`.

`Pandoriclet` provides the same interface as all our anaphoric let variants so we can use it to make our usual counter closure:

```
* (setf (symbol-function 'pantest)
    (pandoriclet ((acc 0))
      (lambda (n) (incf acc n))))

#<Interpreted Function>
```

Which works as expected:

```
* (pantest 3)

3
* (pantest 5)

8
```

However, now we have direct access to the binding that was called `acc` when the closure was created:

```
* (pantest :pandoric-get 'acc)

8
```

And we can similarly change the value of this binding:

```
* (pantest :pandoric-set 'acc 100)

100
* (pantest 3)

103
```

Even the value of our `this` anaphor is accessible, since we deliberately left the anaphor open and added the symbol `this` to the `letargs` binding list when the macro was expanded:

```
* (pantest :pandoric-get 'this)

#<Interpreted Function>
```

So this closure we've created with `pandoriclet` is actually no longer closed. The environment used by this closure—even when all lexical variable symbols have been removed by the compiler—is still accessible through our anonymous function returned from `pandoriclet`. How does this work? With pandoric macros, additional code is compiled in to provide a way to access the closure from outside. But the power of pandoric macros can't be seen by looking at this low-level view of what is happening. What we have done is created an *inter-closure protocol*, or message passing system, for communicating between closures.

Before we continue with pandoric macros, first we need to point out one of the most important examples of *duality of syntax* in COMMON LISP: *generalised variables*. The details of this are complicated, and I won't describe all of them here. For that I recommend Graham's *On Lisp* which has the best treatment I am aware of. The finer points are subtle the idea is simple: accessing a generalised variable is syntactically dual to setting it. You have

Listing 6.19: GET-PANDORIC

```
(declaim (inline get-pandoric))

(defun get-pandoric (box sym)
  (funcall box :pandoric-get sym))

(defsetf get-pandoric (box sym) (val)
  `(progn
     (funcall ,box :pandoric-set ,sym ,val)
     ,val))
```

only one setter form, `setf`, which is capable of setting all types of variables by using the same syntax you would use to access them.

For example, with a regular variable you usually access its value through its symbol, say, `x`. To set it, you can use `(setf x 5)`. Similarly, to access the car slot of a cons called, say, `x`, you use `(car x)`. To set it, you can use the form `(setf (car x) 5)`. This hides the fact that the actual way to set a cons is to use the function `rplaca`. By implementing this duality of syntax we cut in half the number of accessors/setters we need to memorise and, most importantly, enable new ways for us to use macros.

The function `get-pandoric` is a wrapper around the inter-closure protocol getter syntax. It is declaimed inline to eliminate any performance impact caused by this wrapping.

`Defsetf` is an interesting COMMON LISP macro not entirely unlike our `defmacro!` extension to `defmacro` in that it implicitly binds gensyms around provided forms. `Defsetf` works great for defining the setter side of a generalised variable duality as long as the getter can be expressed as a function or macro that evaluates all of its arguments exactly once. Note that while `get-pandoric` could have been defined as a macro, the only reason to do that would be for inlining purposes. Macros are not for inlining, compilers are for inlining.

So going back to our pandoric counter stored in the symbol-function of `pantest`, we can use this new getter function to retrieve the current value of `pantest`'s `acc` binding:

```
* (get-pandoric #'pantest 'acc)

103
```

And now, thanks to generalised variables and `defsetf`, we can use a syntactic dual to set it:

```
* (setf (get-pandoric #'pantest 'acc) -10)

-10
* (pantest 3)

-7
```

The environments that close over functions—what we've been calling the let in *let over lambda*—are starting to look like regularly accessible generalised variables, just like a cons cell or a hash-table entry. Closures are now even more *first-class* data structures than they used to be. Bindings that were previously closed to outside code are now wide open for us to tinker with, even if those bindings were compiled to something efficient and have long since had their accessor symbols forgotten.

But any discussion of generalised variables is incomplete without a mention of its close relative: the *symbol macro*. `Symbol-macrolet`, like its name implies, allows us to expand symbols into general lisp forms. Since it is intuitive and more flexible to use forms that look like function calls to represent macro transformations[20], there isn't much use for `symbol-macrolet` save one important application for which it is vital: symbol macros let us hide generalised variables such that users of our macro think they are accessing regular lexical variables.

The introduction of symbol macros has resulted in one of the strangest *kludges* to the COMMON LISP language: normally when setting a variable accessed through a regular symbol, like `(setf x t)`, `setf` will expand into a `setq` form because this is

[20]Symbol macros take no arguments so a symbol macro definition always expands the same.

Listing 6.20: WITH-PANDORIC

```
(defmacro! with-pandoric (syms o!box &rest body)
  `(symbol-macrolet
     (,@(mapcar #`(,a1 (get-pandoric ,g!box ',a1))
                syms))
     ,@body))
```

what `setq` was originally designed to do: set lexical and dynamic variables (which are always referred to by symbols). But the `setq` special form cannot set generalised variables, so when symbol macros were introduced and it became possible for symbols to represent not just a lexical/dynamic binding but instead any generalised variable, it was necessary to mandate that `setq` forms setting symbols with symbol macro definitions are to be converted back into `setf` forms. Strangely, this really is the *right* thing to do because it allows macros that completely hide the presence of generalised variables from the macro's users, even if they choose to use `setq`. The *really right* solution would be to remove the redundant `setq` from the language in favour of the more general `setf`, but this will not happen for obvious compatibility reasons and because during macro creation `setq` can also be a useful safety shortcut—`setf` plus a check that a symbol has been spliced in instead of a list form. When using `setq` for this remember that it only helps for its splicing safety property; as we've seen, a symbol can reference any generalised variable thanks to `symbol-macrolet`.

The `with-pandoric` macro expands into a `symbol-macrolet` which defines a symbol macro for each symbol provided in `syms`. Each symbol macro will expand references to its symbol in the lexical scope of the `symbol-macrolet` into generalised variable references using our `get-pandoric` accessor/setter to access the result of evaluating the second argument to the macro: `o!box` (stored in `g!box`).

So `with-pandoric` lets us peek into a closure's closed over variable bindings:

```
* (with-pandoric (acc) #'pantest
```

```
    (format t "Value of acc: ~a~%" acc))
Value of acc: -7
NIL
```

As per our design of using generalised variables to form a syntactic dual for the setting and getting of this variable, we can even pretend it is a regular lexical variable and set it with `setq`:

```
* (with-pandoric (acc) #'pantest
    (setq acc 5))

5
* (pantest 1)

6
```

We have now looked at most of the pieces that make up pandoric macros. First, a macro for creating closures: `pandoriclet`, which captures an anaphor, `this`, referring to the actual function used when invoking the closure. This macro also compiles in some special code that intercepts certain invocations of this closure and instead accesses or modifies its closed-over lexical variables. Second, a single syntax for both accessing and setting an accessor is is implemented with `get-pandoric` and `defsetf`. Finally, the macro `with-pandoric` uses `symbol-macrolet` to install these generalised variables as seemingly new lexical variables with the same names as the closed-over variables. These variables refer to the original environment created with `pandoriclet`, but from separate lexical contexts.

As an example, we relate this ability to open up closures by comparing it to the *hotpatching* macros from *section 6.5, Hotpatching Closures.* Recall that `alet-hotpatch` and its closed anaphor cousin, `let-hotpatch`, create closures with an indirection environment so that the function that is called when the closure is invoked can be changed on the fly. The biggest limitation with these macros is that they force you to throw out all the lexical bindings that closed over the previous anonymous function when

Listing 6.21: PANDORIC-HOTPATCH

```
(defun pandoric-hotpatch (box new)
  (with-pandoric (this) box
    (setq this new)))
```

you hotpatched it. This was unavoidable because when we wrote those macros, closures were closed to us.

With **alet-hotpatch** and **let-hotpatch**, we had to compile special purpose code into each closure that was capable of setting the **this** anaphoric lexical binding to its new value. But since we can now open up a closure defined with **pandoriclet** and run this setter code externally, we can define a hotpatching function **pandoric-hotpatch** that will work with any pandoric closure.

Sometimes abstractions just feel right and it is hard to exactly say why. Probably because most programming is the disharmonious combining of disjoint parts, it is surprising and pleasant when you discover abstractions that—seemingly by chance—fit together perfectly. **Pandoric-hotpatch** just seems to read exactly like how it works: it opens a pandoric interface, takes the variable **this** from the lexical scope of the closure **box**, and then uses **setq** to set **this** to the closure to be hotpatched in, **new**.

We can even use **pandoric-hotpatch** on a pandoric closure we created before we knew we wanted it to be hotpatchable. Remember the counter closure we have been playing with throughout this section? It should still be bound to the **symbol-function** of the symbol **pantest**. We were at 6 and counting up when we left off:

```
* (pantest 0)

6
```

Let's install a new closure—one that has a new binding for acc starting at 100, and is counting down:

```
* (pandoric-hotpatch #'pantest
    (let ((acc 100))
```

Listing 6.22: PANDORIC-RECODE

```
(defmacro pandoric-recode (vars box new)
  `(with-pandoric (this ,@vars) ,box
     (setq this ,new)))
```

```
      (lambda (n) (decf acc n))))

#<Interpreted Function>
```

Sure enough, the hotpatch went through:

```
* (pantest 3)

97
```

So now our counter closure has a new value bound to **this** that it uses to perform the counting. However, did this hotpatch change the pandoric value of **acc** binding?

```
* (with-pandoric (acc) #'pantest
    acc)

6
```

No. **Acc** is still the previous value, 6, because the only binding we changed in the pandoric environment was **this**, and we changed that to a new closure with its own binding for **acc**.

The macro **pandoric-recode** takes a slightly different approach to hotpatching. It conserves the original lexical environment of the code while still managing to change the function to be executed when the closure is invoked to something coded and compiled externally. Sound too good to be true? Remembering that the current value for **acc** is 6 in the original pandoric environment, we can use **pandoric-recode** to install a new function that makes use of this original value and, oh, let's say, decrements the counter by half the value of the provided **n**:

Listing 6.23: PLAMBDA

```
(defmacro plambda (largs pargs &rest body)
  (let ((pargs (mapcar #'list pargs)))
    `(let (this self)
       (setq
         this (lambda ,largs ,@body)
         self (dlambda
                (:pandoric-get (sym)
                  ,(pandoriclet-get pargs))
                (:pandoric-set (sym val)
                  ,(pandoriclet-set pargs))
                (t (&rest args)
                  (apply this args)))))))
```

```
* (pandoric-recode (acc) #'pantest
    (lambda (n)
      (decf acc (/ n 2))))

#<Interpreted Function>
```

Sure enough, we have the new behaviour, which decrements acc by `(* 1/2 2)`, from 6 to 5:

```
* (pantest 2)

5
```

And is it associated with the original pandoric binding?

```
* (with-pandoric (acc) #'pantest
    acc)

5
```

Yes. How does `pandoric-recode` work? It closes over the provided lambda form with the pandorically opened bindings of the original closure.

The macro we have used so far to create pandoric closures is `pandoriclet`. `Plambda` is an inside out re-write of `pandoriclet`

that adds a few important features. First and foremost, `plambda` no longer creates the let environment to be used through our pandoric accessors. Instead, `plambda` takes a list of symbols that refer to variables that are expected to be in the caller's lexical environment. `Plambda` can *export* any variables in your lexical environment, transparently making them available for other lexical scopes to access—even ones written and compiled before or after the `plambda` form is.

This is an incremental improvement to our *let over lambda* closure system designed to maximise dual syntax. Thanks to pandoric macros, the most important of which are `plambda` and `with-pandoric`, we can easily and efficiently transcend the boundaries of lexical scope when we need to. Closures are no longer closed; we can open closures as easily as re-writing our lambda forms to be plambda forms. We use `plambda` to export lexical variables and `with-pandoric` to import them as completely equivalent lexical variables. In fact these new variables are so equivalent that they aren't even really new variables at all. A better way of thinking about pandoric variables are that they are simply an extension of the original lexical scope. As a simple example use of `plambda`, here is a pandoric counter that exports variables from two potentially different lexical environments:

```
* (setf (symbol-function 'pantest)
    (let ((a 0))
      (let ((b 1))
        (plambda (n) (a b)
          (incf a n)
          (setq b (* b n))))))

#<Interpreted Function>
```

Notice how easy it was to export these lexical references. Making a closure pandoric is as easy as adding a p character before the `lambda` and adding a list of variables to export after the lambda arguments. We can the open this closure—and any pandoric closure that exports the symbols `a` and `b`—by using `with-pandoric` like so:

Listing 6.24: MAKE-STATS-COUNTER

```
(defun make-stats-counter
       (&key (count 0)
             (sum 0)
             (sum-of-squares 0))
  (plambda (n) (sum count sum-of-squares)
    (incf sum-of-squares (expt n 2))
    (incf sum n)
    (incf count)))
```

```
* (defun pantest-peek ()
    (with-pandoric (a b) #'pantest
      (format t "a=~a, b=~a~%" a b)))

PANTEST-PEEK
* (pantest-peek)

a=0, b=1
NIL
```

Plambda is an example of how factoring out general components of macro expansions can be helpful. Remember when we wrote pandoriclet and decided to move the creation of case statements for the getter code to the function pandoriclet-get and the setter code to pandoriclet-get? Plambda makes use of these same functions. Even though these macros splice the results from these functions into fairly different lexical contexts, since both macros have been written to use the same variable naming convention and inter-closure protocol, the code is re-usable.

So pandoric macros break down lexical boundaries. They allow you to open up closures whenever needed and represent a beautiful confluence of a variety of COMMON LISP language features: anaphoric macros, generalised variables, and symbol macros. But what good are they, really?

Pandoric macros are important because they give us the main advantages of object systems like CLOS without requiring us to depart from the more natural let-lambda combination program-

Listing 6.25: DEFPAN

```
(defmacro defpan (name args &rest body)
  `(defun ,name (self)
     ,(if args
        `(with-pandoric ,args self
           ,@body)
        `(progn ,@body))))
```

ming style. In particular, we can add functionality, or *methods*, for closures to use without having to re-instantiate instances of already created objects.

`Make-stats-counter` is a lambda over let over plambda we have created to create counters, except that it maintains three pieces of information. In addition to the sum, the sum of the squares, and the number of items so far processed are also kept. If we had used `lambda` instead of `plambda` in the definition of `make-stats-counter`, most of this information would be inaccessible to us. We would be locked out because these variables would be closed to us.

How do we write pandoric methods? We can simply access the variables using `with-pandoric` as we have demonstrated above, or, since this is lisp, design a more specific interface.

`Defpan` is a *combination* of the `defun` and `with-pandoric` macros. `Defpan`'s main purpose is to enable a *duality of syntax* between function writing using `defun` and foreign lexical scope access using `with-pandoric`. Although we provide arguments to `defpan` using the same syntax as in lambda forms—a list of symbols—the arguments to `defpan` mean something different. Instead of creating a new lexical environment, these *pandoric functions* extend the lexical environment of the pandoric closures they are applied to. With `defun` and regular lambda forms, the name (symbol) you give a variable is unimportant. With pandoric functions, it is everything. Furthermore, with pandoric functions the order of the arguments doesn't matter and you can elect to use as few or as many of the exported lexical variables as you please.

`Defpan` also provides an anaphor called `self` that allows us to

Listing 6.26: STATS-COUNTER-METHODS

```
(defpan stats-counter-mean (sum count)
  (/ sum count))

(defpan stats-counter-variance
        (sum-of-squares sum count)
  (if (< count 2)
    0
    (/ (- sum-of-squares
          (* sum
             (stats-counter-mean self)))
       (- count 1))))

(defpan stats-counter-stddev ()
  (sqrt (stats-counter-variance self)))
```

perform a useful technique called *anaphor chaining*. By invisibly passing the value of `self` between pandoric functions, we can maintain the value of this anaphor throughout a chain of function calls. As with all chaining constructs, be sure you don't end up in an infinite loop.

Three methods are presented that can be used on our closures created with `make-stats-counter` or any other pandoric closure that exports the necessary variable names. `Stats-counter-mean` simply returns the averaged value of all the values that have been passed to the closure. `Stats-counter-variance` computes the variance of these values by following a link in the chain and `stats-counter-stddev` follows yet another to compute the standard deviation. Notice that each link in the chain only needs to pass on the anaphor `self` to refer to the full lexical context of the closure. We see that the individual pandoric functions only need to reference the variables they actually use and that these variables can be referred to in any order we wish.

So `plambda` creates another anaphor—`self`. While the anaphor `this` refers to the actual closure that is to be invoked, `self` refers to the indirection environment that calls this closure. Although it sounds a bit peculiar, code inside our `plambda` can use `self` to pandorically access its own lexical environment

Listing 6.27: MAKE-NOISY-STATS-COUNTER

```
(defun make-noisy-stats-counter
       (&key (count 0)
             (sum 0)
             (sum-of-squares 0))
  (plambda (n) (sum count sum-of-squares)
    (incf sum-of-squares (expt n 2))
    (incf sum n)
    (incf count)

    (format t
      "~&MEAN=~a~%VAR=~a~%STDDEV=~a~%"
           (stats-counter-mean self)
           (stats-counter-variance self)
           (stats-counter-stddev self))))
```

instead of directly accessing it. This so far only seems useful for **defpan** methods that have been written to work inside our lexical scope.

Make-noisy-stats-counter is identical to **make-stats-counter** except that it uses the **self** anaphor to invoke our **defpan** functions **stats-counter-mean**, **stats-counter-variance**, and **stats-counter-stddev**.

Plambda and **with-pandoric** can re-write lexical scope in any way we please. We conclude this chapter with such an example. A limitation of lexical scope sometimes lamented upon is the fact that the COMMON LISP function **eval** will throw out your current lexical environment when it evaluates the form passed to it. In other words, **eval** evaluates the form in a *null lexical environment.* In COMMON LISP it couldn't be any other way: **eval** is a function. Here is the problem:

```
* (let ((x 1))
    (eval
      '(+ x 1)))

Error: The variable X is unbound.
```

Sometimes it would apparently be desirable to extend your

Listing 6.28: PANDORIC-EVAL

```
(defvar pandoric-eval-tunnel)

(defmacro pandoric-eval (vars expr)
  `(let ((pandoric-eval-tunnel
           (plambda () ,vars t)))
     (eval `(with-pandoric
              ,',vars pandoric-eval-tunnel
              ,,expr))))
```

lexical environment to `eval`. But be careful. Often it is said that if you are using `eval` you are probably doing something wrong. Misuse of `eval` can result in slower programs because `eval` can be a very expensive operation—mostly because it needs to expand macros present in the form passed to it. Should you suddenly find a need for `eval` when programming, ask yourself why you didn't do whatever it is you want to do a lot earlier. If the answer is that you couldn't have, say because you just read the form in, then congratulations, you have found one of the rare legitimate uses of `eval`. Any other answers will lead straight back to the way you probably should have done it in the first place: with a macro.

But let's say that you really do want to `eval` something, if only you could carry along that pesky lexical context. The `pandoric-eval` macro is a fun example use `plambda` and `with-pandoric`. `Pandoric-eval` uses a special variable that we have named `pandoric-eval-tunnel` to make a pandoric closure available to the `eval` function through the dynamic environment. We choose exactly which lexical variables to *tunnel* through the dynamic environment by providing a list of all their symbols as the first argument to `pandoric-eval`. Here it is applied to our earlier example:

```
* (let ((x 1))
    (pandoric-eval (x)
      '(+ 1 x)))
```

```
2
```

And the expression evaluated by `pandoric-eval` can modify the original lexical environment; `pandoric-eval` is a two way tunnel:

```
* (let ((x 1))
    (pandoric-eval (x)
      '(incf x))
    x)

2
```

This section, although very lengthy, has still only scratched the surface of what is possible with pandoric macros and their many possible variations. I am looking forward to the many interesting future developments that will come out of them.

Exercise: Can `pandoric-eval` calls nest? That is, can you use `pandoric-eval` to evaluate a form that evaluates `pandoric-eval`? Why or why not?

Exercise: Although the implementation of pandoric macros here is fairly efficient, it could be improved. Try replacing `pandoriclet-get` and `pandoriclet-set` to generate code that uses a hash-table instead of `case` and benchmark these two implementations for small and large numbers of pandoric variables. Investigate your favourite CLOS implementation, imitate how dispatching is done there, re-benchmark.

Chapter 7

Macro Efficiency Topics

7.1 Lisp Is Fast

> If you tile a floor with tiles the size of your thumbnail, you don't waste many.
>
> —Paul Graham

Some people think lisp is slow. While this may have been true for some early lisp implementations, it has been demonstrably false for many years. In fact, modern lisps like COMMON LISP have been designed to let us use macros to make lisp fast. Really fast. The goal of this chapter might surprise you if you believe in the *performance myth* that says that low-level languages are more efficient than lisp. This chapter aims to illustrate that lisp can be faster than any other programming language and that low-level programming languages like C are actually—because they lack macros—at a performance disadvantage to lisp. Lisp allows us to write code that is more efficient than the code we would be obliged to write in any other language. Especially for large and complicated programs, macros allow the creation of code with definite performance advantages over Blub languages. Sometimes our languages are designed so as to have efficient implementations, but more often they are designed to provide maximum expressive power to the programmer. When we do choose efficiency, lisp is fast. Really fast.

While other languages give you small, square tiles, lisp lets you pick tiles of any size and of any shape. With C, programmers always use a language that is directly tied to the capabilities of a fancy fixnum adder. Aside from procedures and structures, little abstraction is possible in C. By contrast, lisp was not designed around the capabilities and limitations of machines at all.

But surely other languages can be written in less efficient, more convenient ways. After all, Perl allows a programmer to perform miracles with a single dense expression but also provides many upgrade paths for faster code. So what does it mean when we say that lisp allows us to control the efficiency of our abstractions like no other language? As you might have come to expect by now, the answer involves the topic of our book: macros.

Instead of inquiring what makes a program fast, it's better to ask what makes a program slow. This is one of the most researched topics in programming. The root causes can be roughly classified into three broad categories: bad algorithms, bad data-structures, and general code.

All language implementations need good algorithms. An algorithm is a presumably well-researched procedural description of how to execute a programming task. Because the investment required in coming up with an algorithm is so much larger than that of implementing one, the use of algorithms is ubiquitous throughout all of computer science. Somebody has already figured out how, and why, and how quickly, an algorithm works; all you have to do to use an algorithm is translate its *pseudo-code* into something that your system can understand. Because COMMON LISP implementations have typically been well implemented by smart people and continuously improved upon for decades, they generally employ some of the best and quickest algorithms around for most common tasks. For instance, CMUCL uses a tuned heap sort implementation for sorting lists and vectors and the Mersenne Twister 19937 algorithm and its ridiculously large period[1] for generating random numbers[2].

[1] `(1- (expt 2 19937))`

[2] The MT19937 sequence is generated by a linear recursion and is, by itself, not suitable for cryptography.

Good data structures are also necessary for any decent programming language. Data structures are so important that ignoring them will cause any language implementation to slow to a crawl. Optimising data structures essentially comes down to a concept called *locality*. Explaining this concept is easy—data that is accessed most frequently should be the fastest to access. Data structures and locality are so important that they can be observed clearly at almost every level of computing where performance gains have been sought: large sets of CPU registers, memory caches, data-bases, and caching network proxies are a few highlights. Lisp offers a huge set of standard data structures and they are generally implemented very well. Hash tables, linked lists (obviously), vectors with fill pointers, packages with *internable* symbols, and much more is specified, available, and well implemented for COMMON LISP programmers to take advantage of.

If lisp provides such good algorithms and data-structures, how is it even possible that lisp code can be slower than code in other languages? The explanation is based on the most important design decision of lisp: *general code*, a concept otherwise familiar to us as *duality of syntax*. When we write lisp code, we use as many dualities as possible. The very structure of the language encourages us to. Part of the reason why lisp programs are usually so much shorter than Blub programs is because any given piece of lisp code can be used for so much more than can a corresponding piece of Blub code so you can re-use it more often. Coming from a Blub language perspective it can feel unusual to have to *write more to get less*, but this is the important lisp design decision we have been talking about—duality of syntax. The more dualities attached to each expression, the shorter a program seems to be. So does this mean that to achieve or exceed C's performance we need to make our lisp programs as long and dangerous as their corresponding C programs? No. Lisp has macros.

7.2 Macros Make Lisp Fast

This section shows examples of using three types of macros to help create efficient programs: regular macros, read macros, and

a new type of macro introduced here: *compiler macros.*

Macros can be used to control the algorithms, data-structures, type checks, safety checks, optimisation levels of code or portions of code, and much more. We can have safe and general code co-existing in the same program—or even function—as fast and dangerous code. In short, no other language provides an open interface for controlling the compiler in the way that lisp does, and it's all thanks to (what else?) macros. A cursory reading of the ANSI standard seems to suggest that macros and *declarations*, the most direct way to communicate with your compiler, don't work well together:

> Macro forms cannot expand into declarations; declare expressions must appear as actual subexpressions of the form to which they refer.

What ANSI means is that the following macro will not work as desired:

```
(defmacro go-fast () ; Broken!
  '(declare (optimize (speed 3) (safety 0))))
```

We can't put macro invocations in places where declarations are expected. Another way to think about this is that the system's code-walker is not required to expand macros in the bodies of special forms before it checks for declarations. Wanting to go fast is such a common thing to declare so maybe we can do even better than the flawed `go-fast` macro above. When we want to compress meaning as much as possible, often we want a *read macro.* Read macros are also suitable for expanding into declarations because they are expanded long before the code-walker attempts to walk the code. They read in as actual subexpressions.

Sharp-f is a read macro that can be used to control the most important performance trade-off available to COMMON LISP programs: the balance between the `speed` and the `safety` declarations. For instance, sharp-f by itself reads in as what we would have liked `go-fast` to expand into:

Listing 7.1: SHARP-F

```
(set-dispatch-macro-character #\# #\f
   (lambda (stream sub-char numarg)
     (declare (ignore stream sub-char))
     (setq numarg (or numarg 3))
     (unless (<= numarg 3)
       (error "Bad value for #f: ~a" numarg))
     `(declare (optimize (speed ,numarg)
                         (safety ,(- 3 numarg))))))
```

```
* '#f

(DECLARE (OPTIMIZE (SPEED 3) (SAFETY 0)))
```

However, we can change this and declare that we value safety above speed by providing a number less than 3 as the reader number argument. All dispatch read macros can accept a numeric argument like this. It is passed as the third argument, often called **numarg**, to the read macro function. Here is an example where we value safety over speed by providing 0:

```
* '#0f

(DECLARE (OPTIMIZE (SPEED 0) (SAFETY 3)))
```

Values of 1 and 2 can also be passed resulting in the following declarations. The advantages of these different declaration settings are very compiler dependent so you will almost never use them:

```
* '(#1f #2F)

((DECLARE (OPTIMIZE (SPEED 1) (SAFETY 2)))
 (DECLARE (OPTIMIZE (SPEED 2) (SAFETY 1))))
```

Although macros can't directly expand into declarations, we can still use regular macros to control declarations. Because the code-walker can't walk a macro form to search for declarations

Listing 7.2: FAST-PROGN

```
(defmacro fast-progn (&rest body)
  `(locally #f ,@body))
```

Listing 7.3: SAFE-PROGN

```
(defmacro safe-progn (&rest body)
  `(locally #0f ,@body))
```

until it has expanded that macro, there is no way that it can tell if this declaration was an actual subexpression of a form you wrote or if the macro added the declaration when it was expanded.

Fast-progn and **safe-progn** are simple examples of macros that expand into forms containing declarations. Notice that we use **locally**'s implicit progn instead of **progn** itself because **progn** doesn't accept declarations[3]. These two macros use the sharp-f read macro we defined previously. We can use these forms as a version of **progn** where its enclosed expressions are optimised for speed (but are dangerous) and a version where they are ensured safe (and possibly slow):

```
* (macroexpand
    '(fast-progn
       (+ 1 2)))

(LOCALLY
  (DECLARE (OPTIMIZE (SPEED 3) (SAFETY 0)))
  (+ 1 2))
T
```

We can also provide other declarations in the macro arguments because their location is not and cannot be verified until the macro is expanded:

```
* (macroexpand
```

[3]Because it doesn't establish bindings.

```
    '(fast-progn
       (declare (type fixnum a))
       (the fixnum (+ a 1))))

(LOCALLY
  (DECLARE (OPTIMIZE (SPEED 3) (SAFETY 0)))
  (DECLARE (TYPE FIXNUM A))
  (THE FIXNUM (+ A 1)))
T
```

When experimenting with macro expansions, sometimes we would like to see what happens when we embed them into various lexical contexts. Combining the read-time evaluation macro described in *section 4.1, Run-Time at Read-Time* with the * variables that keep the last three REPL results available lets us see that our form evaluates as expected:

```
* (let ((a 0))
    #.*)

1
```

But notice that although the above evaluated correctly, declarations are sometimes only fully considered for compiled code. For example, since our evaluation above interpreted the code[4], it will probably ignore the safety declaration and go ahead to promote an overflowed result to a bignum. Let's see if this happens here:

```
* (let ((a most-positive-fixnum))
    #.**)

536870912
```

It does. CMUCL ignores declarations for interpreted code. We want to continue playing with our expression currently in ***, but since we're not sure if we're going to get it on this next try, let's bring it back to * so we don't lose it:

[4]In most implementations.

```
* ***

(LOCALLY
  (DECLARE (OPTIMIZE (SPEED 3) (SAFETY 0)))
  (DECLARE (TYPE FIXNUM A))
  (THE FIXNUM (+ A 1)))
```

There it is. So we now have three more chances to make it work. Let's try compiling it to see if we can observe fixnum wrapping:

```
* (funcall
    (compile nil
      '(lambda ()
         (let ((a most-positive-fixnum))
           ,*))))

; Warning: This is not a (VALUES FIXNUM &REST T):
;   536870912

536870912
```

Hm, what gives? Didn't we tell lisp to not check for this? Reasoning about declarations is further complicated by compile-time optimisations like *constant folding*. What happened was that when lisp compiled the form it was able to perform the addition at compile-time since we are adding constants and therefore it knows the result will also be constant and there is no need to calculate it at run-time. When lisp did this it saw that our declaration for a being a fixnum will definitely be wrong. The warning is lisp's way of telling us "You dummy, I'm ignoring your declaration because you can't be trusted." If we slightly shuffle the expression around so that lisp can't fold any constants, we can finally see the fixnum wrapping in effect:

```
* (funcall
    (compile nil
      '(lambda (a)
```

```
        ,**))
    most-positive-fixnum)

-536870912
```

Another important property of declarations is that they can *shadow* other declarations in the same sense that lexical variables can shadow other lexical variables. For instance, we might like to write a macro that performs safety checks even when it is embedded in code declared unsafe:

```
(defmacro error-checker ()
  `(safe-progn
     (declare (type integer var))
     do-whatever-other-error-checking))
```

Wrapping one more layer, we can use these macros to add error checking code around some code that needs to run quickly instead of safely by *nesting* another use of the other of these macros, `fast-progn`:

```
(defun wrapped-operation ()
  (safe-progn
    do-whatever-error-checking
    (fast-progn
      but-this-needs-to-go-fast)))
```

Safely verifying parameters with an error checking region surrounding a fast implementation of some functionality is a common pattern in high performance lisp code. Especially for iterative procedures like array traversal, dramatic improvements to run-time performance can be achieved by doing error checks like type and bounds checking up front at the start of an operation, then omitting them where possible while performing it.

COMMON LISP is first and foremost designed to be powerful to program; efficiency is a distant secondary concern. However, these features, power and efficiency, don't necessarily represent a trade-off. With macros we can apply the power of lisp

to solving efficiency problems. In addition to regular macros and read macros—which by themselves already provide considerable power—COMMON LISP also provides *compiler macros*. Compiler macros are macros in the same sense that the other types of macros are: they are programs that program. Compiler macros aren't well described in most lisp tutorials which is indicative of just how often performance is a priority to programmers (almost never). However, compiler macros are elegant solutions to certain classes of efficiency problems and deserve to be in every lisp professional's tool-kit.

Compiler macros define transformations that your lisp compiler will apply to (named) function invocations. That means that you can take a function created with `defun` and tell lisp that instead of compiling calls to this function, it should instead compile some arbitrary code indicated by the compiler macro. Why use a function in combination with a compiler macro instead of just writing a macro with that name in the first place? The first, less important reason is that this allows us more control over when to absorb the overhead of compilation. In particular, COMMON LISP doesn't specify when or how often a macro will be expanded. In interpreted code it is possible that a macro will be expanded every single time it is invoked[5]. When doing compile-time optimisation we want to perform a (presumably lengthy and expensive) calculation before running the function so as to reduce the amount of calculation the function itself has to do. Compiler macros give us a means of performing a lengthy compilation calculation exactly once, when we compile the code—as it should be.

But more importantly than performing the compilation calculation only once and at the right time, compiler macros are useful because they introduce *dualities of syntax* into the language. Compiler macros let us add a dual meaning to any form representing a (named) function call. In addition to its regular meaning, a compiler macro adds a compiled meaning. You are very strongly advised to ensure that your compiled meaning implements the same task as your regular meaning, but you are free

[5]In other words, caching of macro expansions isn't guaranteed when interpreting.

to vary how it performs it (that's the point). The advantage of using dual syntax is that we can make changes to the efficiency of code without having to modify that code at all. We can take an existing code base—one that presumably uses lots of function calls—and change how that code is compiled by introducing dual syntax. All we have to do is find the function invocations that are unacceptably costly and then implement compiler macros to convert them into cheaper expansions.

What sort of function invocations are costly? As a first example, recall from *section 4.6, Reader Security* that functions can perform *lambda destructuring* and that this is a subset of the more general *defmacro destructuring*[6]. When functions accept keyword arguments we pass them as grouped pairs of keyword symbols and their corresponding values. Keyword arguments are extremely useful but functions that use them unfortunately suffer more invocation overhead than functions that don't. Destructuring isn't free. The compiler needs to compile code into the function that scans across a necessarily variable length argument list, gets the values into the right order (including inserting default values), and then actually executes the function. Generally, lisp compiles very fast code for destructuring these keyword arguments and so we almost never notice (or care about) this minor inefficiency. However, there are situations when we do care, particularly when we call such a function from within a performance-critical loop.

`Fast-keys-strip` is a utility that takes a lambda destructuring list consisting of nothing but regular and keyword arguments and returns a list of the symbols used to refer to these arguments. In other words, it returns `(a b c)` when passed `(a b c)` or `(a &key b (c 0))`, but passing it `(a &optional b c)` is forbidden.

`Defun-with-fast-keywords` is used in the same way as `defun`. Like `defun`, its first argument is a symbol naming the function, the second an argument list, and the rest forms for the function being defined to execute. Unlike `defun`, however, `defun-with-fast-keywords` forms can only be given regular and keyword arguments

[6]Lambda destructuring can't destructure lists passed as arguments and lacks some of the defmacro features like whole.

Listing 7.4: FAST-KEYWORDS-STRIP

```
(defun fast-keywords-strip (args)
  (if args
    (cond
      ((eq (car args) '&key)
        (fast-keywords-strip (cdr args)))
      ((consp (car args))
        (cons (caar args)
              #1=(fast-keywords-strip
                   (cdr args))))
      (t
        (cons (car args) #1#)))))
```

Listing 7.5: DEFUN-WITH-FAST-KEYWORDS

```
(defmacro! defun-with-fast-keywords
           (name args &rest body)
  `(progn
     (defun ,name ,args ,@body)
     (defun ,g!fast-fun
            ,(fast-keywords-strip args)
            ,@body)
     (compile ',g!fast-fun)
     (define-compiler-macro ,name (&rest ,g!rest)
       (destructuring-bind ,args ,g!rest
         (list ',g!fast-fun
               ,@(fast-keywords-strip args))))))
```

Listing 7.6: KEYWORDS-TESTS

```
(defun
  slow-keywords-test (a b &key (c 0) (d 0))
  (+ a b c d))

(compile 'slow-keywords-test)

(defun-with-fast-keywords
  fast-keywords-test (a b &key (c 0) (d 0))
  (+ a b c d))
```

(no optionals, rests, etc). Exercise: Extend `fast-keywords-strip` to handle all lambda destructuring lists[7].

The expansion of `defun-with-fast-keywords` is complicated. It expands into three forms[8]. The first defines the function just as if we had used regular `defun` instead. The second defines a function named with an automatic gensym, `g!fast-fun`. This function is similar to the first, except that it simply takes a non-keyword argument for every argument (keyword or not). Next a compiler macro is defined to transform calls to our first function into calls to the second function. So instead of having the first function perform the keyword destructuring, we take advantage of the compile-time knowledge of the form used to invoke the function and put the keywords together into the right order with destructuring bind.

We now have a (nearly) dual syntax with `defun`. A regular definition of a function with keyword arguments looks like `slow-keywords-test`. It is compiled for benchmarking purposes below. `Fast-keywords-test` is written identically to `slow-keywords-test` except it uses `defun-with-fast-keywords` instead of `defun`. It turns out we don't need to compile this function because `defun-with-fast-keywords` expands into a call to compile on the only one of its definitions that needs it—the one named by the

[7]But keep in mind Norvig's golden law of lisp: never mix keyword and optional arguments.

[8]Which are all treated as top-level because top-level progn forms are treated specially—a valuable feature of COMMON LISP.

Listing 7.7: KEYWORDS-BENCHMARK

```
(defun keywords-benchmark (n)
  (format t "Slow keys:~%")
  (time
    (loop for i from 1 to n do
      (slow-keywords-test 1 2 :d 3 :c n)))
  (format t "Fast keys:~%")
  (time
    (loop for i from 1 to n do
      (fast-keywords-test 1 2 :d 3 :c n))))

(compile 'keywords-benchmark)
```

automatic gensym **g!fast-fun**.

Keywords-benchmark is a simple function that uses the **time** macro to tell us how long an equivalent series of calls takes against both the functions. Notice that we compile **keywords-benchmark** too. More about benchmarking will be said in *section 7.7, Writing and Benchmarking Compilers.*

```
* (keywords-benchmark 100000000)
Slow keys:
; Evaluation took:
;   17.68 seconds of real time

Fast keys:
; Evaluation took:
;   10.03 seconds of real time
```

Calling this function 100 million times is enough for us to see that even when both the functions are compiled, the function defined with **defun-with-fast-keywords** runs about 40% faster thanks to its compiler macro. Notice also that our compiler macro's performance doesn't rely on having the keyword arguments be constant values known at compile-time. See that we pass **n**, a varying lisp form, as the argument to the **:c** keyword. So the compiler macro expands the fast version into an identical

version as the slow one except that it doesn't have the keyword destructuring overhead.

So why doesn't COMMON LISP do this for every function that accepts keywords and always avoid the overhead? Compiler macros are only applied at compile-time but we want to retain the ability to destructure arguments at run-time. Here is the important point about compiler macros: compiler macros are optimisations to function invocations, not to the functions themselves. In the case of keywords, compiler macros allow us to eliminate the overhead for compiled invocations of a function while still leaving the original function—and its keyword destructuring code—available for use at run-time. Compiler macros give us a dual syntax for two different operations which are distinguishable only by context. For another way to avoid keyword overhead see Norvig's PAIP[PAIP-P323].

What other function invocations can benefit from compiler macros? Not only can we reduce destructuring overhead, but often we can reduce the overhead of the function itself by pre-processing constant arguments. A compiler macro can perform some preparation at compile-time so it doesn't have to be done at run-time. One of the most obvious examples of this is the `format` function. Consider how `format` (or, in C, `printf`) works. It is a function that you pass a control string to at run-time. `Format` then processes the control string and prints the formatted output to a stream (or returns it as a string). In essence, when you use `format` you make a function call to a format string interpreter with the control string being the program. With compiler macros we can eliminate the function call, pre-process the control string, and change the function invocation to specialised code spliced into the call-site where the compiler can make further optimisations. Sounds difficult, doesn't it? We have to know how to convert format control strings into their equivalent lisp code. Luckily, as with so many other things, COMMON LISP has already thought about this. COMMON LISP does formatting *right*. That is the domain specific language that it specifies for creating formatted output can macro-compile itself down to lisp code. This is part of lisp philosophy—everything should compile down to lisp. The

macro that compiles control strings to lisp is **formatter**. When you give a control string to **formatter** it will expand into a lambda form that performs the desired formatting. For example, here is an expansion for a simple control string[9]:

```
* (macroexpand '(formatter "Hello ~a~%"))

#'(LAMBDA (STREAM &OPTIONAL
                  (#:FORMAT-ARG-1783
                    (ERROR "Missing arg"))
                  &REST FORMAT::ARGS)
    (BLOCK NIL
      (WRITE-STRING "Hello " STREAM)
      (PRINC #:FORMAT-ARG-1783 STREAM)
      (TERPRI STREAM))
    FORMAT::ARGS)
T
```

So **formatter** expands into a lambda form[10]. It has compiled the control string into a lisp form, suitable for evaluation or for macro embedding into other lisp code where it will become a compiled function or will be inlined into compiled code at the call-site. Notice however that the expansion of **formatter** must be passed a stream and cannot accept **nil** like **format** can. This is because the functions that **formatter** expands into (like **write-string** and **terpri**) require streams. Use the **with-output-to-string** macro to get around this.

Fformat is a perfectly transparent wrapper around **format**. It exists so we can define a compiler macro for formatting. We need a new function name from **format** because defining compiler macros over top of functions specified by COMMON LISP is forbidden[CLTL2-P260]. Our compiler macro takes advantage of a defmacro destructuring feature, &whole. We use this to bind **form** to the actual list structure of the macro invocation. This is done to take advantage of a feature of compiler macros: compiler

[9] **Terpri** prints a newline character to the stream.

[10] A sharp-quoted lambda form to be exact.

Listing 7.8: FFORMAT

```
(defun fformat (&rest all)
  (apply #'format all))

(compile 'fformat)

(define-compiler-macro fformat
                       (&whole form
                        stream fmt &rest args)
  (if (constantp fmt)
    (if stream
      `(funcall (formatter ,fmt)
         ,stream ,@args)
      (let ((g!stream (gensym "stream")))
        `(with-output-to-string (,g!stream)
           (funcall (formatter ,fmt)
             ,g!stream ,@args))))
    form))
```

macros can choose to not expand at all. If we return `form`, lisp will see that we are simply returning the form we were passed (by checking it with `eq`) and will ask the compiler macro for no further expansions of the form—even though we are expanding into a use of a function with a compiler macro. While compiling we have elected to use the other meaning of the form. This is the fundamental difference between a compiler macro and a regular macro. A compiler macro can share an exact dual syntax with a function but a regular macro cannot. In `fformat` a compiler macro will elect to not expand into the more efficient meaning when its control string argument is non-constant. In `fformat`, we still want invocations of `fformat` on non-string control strings (like function calls that return strings) to work. In other words, we still want to be able to generate control strings at run-time. Such invocations obviously cannot use compile-time optimisations on the control strings.

`Fformat-benchmark` is nearly identical to the `keywords-benchmark` function presented earlier. It uses `time` to compare the time required to perform a large number of format operations

Listing 7.9: FFORMAT-BENCHMARK

```
(defun fformat-benchmark (n)
  (format t "Format:~%")
  (time
    (loop for i from 1 to n do
      ( format nil "Hello ~a ~a~%" 'world n)))
  (format t "Fformat:~%")
  (time
    (loop for i from 1 to n do
      (fformat nil "Hello ~a ~a~%" 'world n))))

(compile 'fformat-benchmark)
```

using both regular **format** and the new **fformat**. Here are the results for a million iterations:

```
* (fformat-benchmark 1000000)
Format:
; Evaluation took:
;   37.74 seconds of real time
;   [Run times include 4.08 seconds GC run time]
;   1,672,008,896 bytes consed.

Fformat:
; Evaluation took:
;   26.79 seconds of real time
;   [Run times include 3.47 seconds GC run time]
;   1,408,007,552 bytes consed.
```

About a 30% improvement. Not only does the compiler macro decrease the time required to perform the formatting, but it also conses less (which in turn decreases its garbage collection time). The compiler macro has avoided the interpretation of the format string at run-time, instead performing most of the calculation only once when the function was compiled—as it should be. Unfortunately, benchmarks often obscure or eliminate important details. While pre-compiling format strings with **fformat** will eliminate the interpretation overhead, it will do so at the cost of a larger

compiled program. Even if main memory is plentiful, large code can run more slowly by reducing instruction cache performance.

In this section we have looked at ways to customise the performance of code using regular macros, read macros, and a special type of macro designed just for this task: compiler macros. Hopefully this section and the remainder of this chapter will convince you that if you want to write really efficient code, you want COMMON LISP. And you want COMMON LISP because of macros.

Exercise: Download Edi Weitz's CL-PPCRE (described in *section 4.4, CL-PPCRE*) and see how `api.lisp` uses compiler macros. Visit Edi's website and download a few of his lisp packages that seem interesting.

Exercise: When we wrote the compiler macro for `fformat` we were forced to use `gensym` explicitly because there is no `define-compiler-macro!` macro. Fix this. Difficult Exercise: Define `define-compiler-macro!` so it uses the functionality of `defmacro!` and doesn't call `gensym` itself. Hint: Think outside the box.

7.3 Getting To Know Your Disassembler

It can be difficult to really get a sense for what is expensive in lisp without inspecting the raw instructions executed by your processor for different lisp forms. Just like when writing macros it often helps to view their expansions, sometimes looking at the *compiled expansions* of lisp programs—usually assembly instructions—is similarly useful. Because lisp compilers can be and often are thought of as macro expanders, the machine code that they generate is, in a strange sort of sense, itself lisp code. Because lisp is not so much a language as it is a building material and fabric for creating languages, lisp is used to define and compile a language that just happens to be the same language of the instruction set of your processor.

COMMON LISP provides us a function called `disassemble` to look at compiled expansions. `Disassemble` is the analog of the CMUCL `macroexpand-all` extension described in [USEFUL-LISP-ALGOS2]. By passing `disassemble` a function, or a symbol for which

a `symbol-function` binding exists, we can look at the raw machine code instructions that will be executed if the function is invoked.

The problem is that these raw machine code instructions don't look at all like lisp. Instead of lisp's soothing nested parenthesis, these instructions are usually strange, tiny steps for some very arbitrary machine. Looking at the compiled expansion of lisp code is similar to reading a poster with a magnifying glass. You can see any portion you like in great detail, but interpreting the *big picture* from this alone can be difficult or impossible. Worse, when looking at code in this level of detail, it can sometimes be impossible to look at any one piece of machine code and tell for certain why the compiler put it there.

Unfortunately, nobody really knows how to best implement lisp past the function of `compile`. There is a lot more macroexpansion to be done to the code, that is certain, some of which so certain that it could probably be standardised, but the best way to use hardware resources like CPU cycles and memory is still (and might always be) a very hot research topic. Even harder to track than improvements in compiler design are the constant improvements to hardware. Optimisations that made sense initially may become irrelevant or even plain incorrect. We don't need to look far for examples of how a changing world affects assumptions about efficiency.

Scientists[11] used to avoid using floating point computations in code that needed to perform well, instead opting for machine-word based, fixed point computations. This was because computers didn't have dedicated floating point hardware and were forced to use the processor's integer instructions to simulate it. Because the processors weren't really optimised for this, floating point operations were always much slower than fixed point operations. However, over time, hardware started to sprout dedicated floating point co-processors which were designed to perform these floating point operations at lightning speed. Almost over-night, scientists went from being able to assume that a fixed point operation would always be much faster than a floating point operation,

[11] One of the very few computer user demographics that ever requires efficient code.

Listing 7.10: DIS

```
(defmacro dis (args &rest body)
  `(disassemble
     (compile nil
       (lambda ,(mapcar (lambda (a)
                          (if (consp a)
                            (cadr a)
                            a))
                        args)
         (declare
           ,@(mapcar
               #'(type ,(car a1) ,(cadr a1))
               (remove-if-not #'consp args)))
         ,@body))))
```

to having to investigate and benchmark their hardware before deciding. Hardware developments changed the performance reality of floating point. Soon after, computers started to come with 2, 4, or more floating point co-processors and scientists discovered that if they were able to keep the *pipeline* of floating point instructions full, floating point operations could often perform even better than fixed point operations. Many programs that chose fixed point for performance reasons went—in the time-frame of a decade or so—from choosing the *right* implementation to choosing the *wrong* implementation.

Just as it is useful to look at the `macroexpand` and `macroexpand-all` output while developing macros, it is helpful to look at the `disassemble` output, not only to learn about how your implementation functions, but also to make sure you are giving lisp all the information it needs to generate efficient expansions. `Dis` is a macro that makes it easy to check at the `disassemble` output for a portion of lisp code. Its first argument is a list of symbols or lists of a type and a symbol. To see how `dis` words, expand it. Here is what `dis` expands to for a simple binary addition:

```
* (macroexpand
    '(dis (a b) (+ a b)))
```

```
(DISASSEMBLE
  (COMPILE NIL
    (LAMBDA (A B)
      (DECLARE)
      (+ A B))))
T
```

Why is the empty `declare` form there? It is a place-holder where `dis` can insert type declarations if you specify them in the parameters like so:

```
* (macroexpand
    '(dis ((fixnum a) (integer b))
       (+ a b)))

(DISASSEMBLE
  (COMPILE NIL
    (LAMBDA (A B)
      (DECLARE (TYPE FIXNUM A)
               (TYPE INTEGER B))
      (+ A B))))
T
```

Because `dis` expands into a (wrapped) lambda form, it works a lot like one. You can add extra declarations if you like, and the return value is important (because lambda forms provide an implicit progn). With this book's code loaded, try entering this form into your lisp environment:

```
(dis (a b)
  (+ a b))
```

The machine code should be fairly short, but this is because much of the complexity is hidden from view with the invocation of a pre-compiled function—one with enough smarts to provide all the fancy lisp number features like type contagion, rational simplification, etc. This is called *indirection* and is probably fairly obvious in your disassembler output:

```
CALL #x1000148 ; GENERIC-+
```

Try it with three arguments:

```
(dis (a b c)
  (+ a b c))
```

Exercise: How many indirections to the generic addition function do you see? How about with `N` where `(<= 0 N)` arguments?

Now try locking down a type for one of the variables. Compare this closely with the earlier example where no types were declared:

```
(dis ((fixnum a) b)
  (+ a b))
```

Some sort of `OBJECT-NOT-FIXNUM-ERROR` should now be apparent. Lisp has compiled in some extra code to do this type checking while still indirecting control to the generic addition function because what the type of `b` is unknown at compile-time and thus might require all of lisp's fancy numerical behaviour like contagion.

This is not how to get fast code. In fact this code might even be slightly less efficient than the previous. For fast code, we need to take advantage of a process called *inlining*. For some special operations, when enough type information is available, lisp compilers know how to avoid indirection and directly add machine code into the function being compiled to perform the desired operation. There should be no indirection to a generic addition function in the following:

```
(dis ((fixnum a) (fixnum b))
  (+ a b))
```

This inlining process probably resulted in more machine code than the one that employed indirection. This is because some (but not all) of the functionality that was implemented in the generic addition function was copied into the function that we compiled. Although it appears longer, in some circumstances this code will perform better because of less indirection.

But this mess of machine code is still a lot less efficient than might be possible in a C implementation. There are still all sorts of argument count, type, and overflow checks compiled in—so many that the actual cost of the addition is still low compared to its overhead. If we used this function in a loop, this overhead might be unacceptable.

With languages like C you specify types everywhere and enforce safety nowhere so code is always efficient, never safe, and always annoying to write. With most dynamic Blub languages you specify types nowhere and enforce safety everywhere so code is always safe, never annoying, but also never efficient. With most strong, static Blub languages you specify types everywhere and enforce safety everywhere so code is always efficient and always safe, but always annoying. Lisp gives you choice. Because lisp defaults to safe-and-not-annoying mode, lisp programs often seem slightly slower than their C equivalents, but are almost always safer. Because lisp allows programmers an excellent type declaration system and implementations have such excellent compilers, lisp programs are almost always just as safe as their dynamic Blub equivalents, and usually a whole lot faster. Most importantly, lisp has macros so if something is annoying, well, change it!

Let's go ahead and ask lisp to make our addition fast. Recall that sharp-f is a read macro abbreviation for a high-speed, low-safety declaration.

```
(dis ((fixnum a) (fixnum b))
  #f
  (+ a b))
```

This sequence of machine code instructions should be a bit shorter than before. The type checks and argument count checks should be removed. But this still isn't quite the single instruction, bang-off, dangerous fixnum addition we were looking for. To get some insight into what is happening, we should check for compiler *notes*. A note is an observation made by the compiler that essentially says: "You look like you're trying to do something efficient, and you're almost there, but I need a little clarification on your intent. Here's a tip for making it more clear..."

Compiler notes are invaluable sources of information. When trying to create efficient lisp code, you should read and consider them carefully. Lisp compilers use *type inference* systems to discover intricate properties of code that even you, the programmer, might not have considered. In the above example, our compiler should somewhere give us a note like:

```
; Note: Doing signed word to integer coercion
;       (cost 20) to "<return value>".
```

Lisp doesn't do anything silly like ignore a fixnum overflow unless we explicitly ask it to[12]. So in order to get lisp to put caution to the wind and write us a function that really burns, but is possibly unsafe, we need to avoid the signed word (fixnum) to integer (bignum) check and coercion. We need to tell lisp that overflows are acceptable and, yes, we really want to silently return a fixnum:

```
(dis ((fixnum a) (fixnum b))
  #f
  (the fixnum (+ a b)))
```

Now we're burning. This is roughly equivalent to a C fixnum addition function: a few machine instructions that add together two registers and then return control to the caller. While your disassembler can offer many insights into all areas of lisp efficiency, there are main two skills that it will teach you. The first skill was mostly covered in this section: how to use declarations to get efficient numerical behaviour, especially inside of loops. The second is how to efficiently use array/vector data-structures. This will be discussed in *section 7.4, Pointer Scope.*

Just like technology advancements that changed floating point arithmetic's efficiency reality from something that should be avoided to something that should be exploited, advancements in lisp compiler technology—combined with COMMON LISP's *right* type and safety declaration systems—are changing how we

[12]In C programs, fixnum overflows are a security vulnerability class that is frequently targeted and exploited by attackers.

think about efficiency. With these tools, and the growing complexity demands of software systems[13], the question is changing from how to make lisp as efficient as low-level languages to how to make other languages as efficient as lisp. The answer, of course, is to use macros to implement them on lisp.

7.4 Pointer Scope

Does removing pointers from a language reduce the power of the language? In particular, does lisp's lack of explicit *pointer scope* prevent us from efficiently implementing algorithms specified in terms of pointer arithmetic? It turns out no, lack of direct support for pointers in lisp does not pose a theoretical or practical challenge. Any algorithm or data structure that can be implemented with pointers in a language like C can be implemented the same or better in lisp.

But, really, what is pointer scope and why might we want to use it? Pointer scope involves treating the computer's memory (or virtual memory) as a large, indexable array from which it can load and store fixnum values. Does this sound dangerous? It should, because it is the source of many complex bugs and the direct cause of several of the largest classes of software security problems today.

Pointer scoping is really a way of specifying indirections, that is accesses across environments, that just also happens to be tied to fixnum arithmetic. How do we normally program across environments? We use either the lexical or dynamic scoping provided by COMMON LISP, dual combinations of the two, or new types of scoping created by macros. The `pointer-&` macro and the `pointer-*` function are examples that sketch some of the illusion of pointer scope for us, showing that when you think you need a pointer, you probably really need a closure. The first and only mention of this analogy between pointers and closures I have heard described was in a post to the `comp.lang.scheme` newsgroup from

[13]Lisp's optimisation potential really shines when macro techniques are used to improve the performance of large and complex applications.

Listing 7.11: POINTER-SCOPING-SKETCH

```
(defmacro! pointer-& (obj)
  `(lambda (&optional (,g!set ',g!temp))
     (if (eq ,g!set ',g!temp)
       ,obj
       (setf ,obj ,g!set))))

(defun pointer-* (addr)
  (funcall addr))

(defsetf pointer-* (addr) (val)
  `(funcall ,addr ,val))

(defsetf pointer-& (addr) (val)
  `(setf (pointer-* ,addr) ,val))
```

Oleg Kiselyov[POINTERS-AS-CLOSURES]. He suggested using closures to emulate pointers and provided an implementation for Scheme[14].

`Pointer-&` and `pointer-*` show a possible way to mimic pointer indirection through closures. When we use the `pointer-&` macro it expands into a lambda form that has some smarts in it to determine if you would like to get or set the value, and does accordingly. `Pointer-&` uses *gensyms* to do this. Instead of using them as a name for a binding so that unwanted variable capture is avoided at compile-time, `pointer-&` uses them to ensure that there is no possible *run-time capture*, where we are prevented from setting a closure's value a certain value because it collides with our implementation. For instance, we might have chosen the lisp default of `nil` for this which would usually work except if we tried to pass `nil` as an argument. Gensyms are convenient to use at run-time because we know there will never be another value `eq` to a gensym. That is their raison-d'etre.

`Pointer-*` and its `defsetf` are the framework for accessing these indirected values through generalised variables. The `defsetf` for `pointer-&` is there so that expansions of `pointer-&`

[14]Oleg's website contains many such insights and is highly recommended reading.

will know how to set nested indirections. As a simple example, we can create a closure that mimics the *pointer to a pointer* pattern common in C by creating a reference to a binding in a let environment:

```
* (let ((x 0))
    (pointer-& (pointer-& x)))

#<Interpreted Function>
```

Let's store this closure for later use by transferring it from the * special variable (let's keep all these asterisks straight):

```
* (defvar temp-pointer *)

#<Interpreted Function>
```

Now we can *dereference* this closure:

```
* (pointer-* temp-pointer)

#<Interpreted Function>
```

It appears we have another closure. We have only dereferenced one step of the pointer chain. Using the * special variable to refer to the previous result, let's dereference further:

```
* (pointer-* *)

0
```

The 0 is the original object that we pointed to. We can also use this dereferencing syntax—which is of course an illusion over closures—to set the value of this object through the pointer chain:

```
* (setf (pointer-* (pointer-* temp-pointer)) 5)

5
```

Sure enough, this changes the original let environment being pointed to so that it has a new value, 5:

```
* (pointer-* (pointer-* temp-pointer))

5
```

If we like, we can add another layer of indirection:

```
* (pointer-& temp-pointer)

#<Interpreted Function>
```

Which now requires three dereferences:

```
* (pointer-* (pointer-* (pointer-* *)))

5
```

And itself can be accessed as a generalised variable:

```
* (setf (pointer-* (pointer-* (pointer-* **))) 9)

9
```

Even though they may be at different levels of indirection, all of the closures in this dereferencing chain still point back to the original let environment:

```
* (pointer-* (pointer-* temp-pointer))

9
```

But this probably isn't what you thought we meant by pointer scope. Because most computer processors consider memory a big array of fixnums, and because C was designed around the capabilities of existing processors, C's pointer scoping is permanently tied to fixnum arithmetic. In C, when you dereference a pointer you always know exactly what is happening: the compiler compiles

in code to index into memory with a fixnum and retrieve or set a fixnum value. The largest difference between C's pointer scoping and our above closure dereferencing technique is that while C allows us to change where a pointer is pointing by adding or subtracting fixnums to it, the closures compiled by `pointer-&` and accessed with `pointer-*` are fixed. The code to access and set them—whatever that might be—is added to the indirection environment at compile time. Even in our simple example above, we used at least two different types of closures, both of which, thanks to generalised variables, were accessible through a uniform dereferencing syntax. The `x` we originally referred to was a lexical variable and the `temp-pointer` *tunnel* variable that we pointed to was a dynamic variable. As we saw in *section 6.7, Pandoric Macros*, we can customise closures, and thus indirection, in any way we want.

So closures are actually more flexible and less dangerous than C style pointers. When you think you need a pointer, you probably need a closure. Rather than just being a fixnum that can be used as an address, closures are code that is compiled to retrieve and set any sort of data in any sort of environment. Although for most tasks closures are the best construct to accomplish indirection, sometimes we would like to take advantage of our processor's fixnum addressing functionality to achieve extremely efficient code. C lets us do it; COMMON LISP lets us do it better.

Using C-style pointers in lisp is actually very straightforward and doesn't require a departure from our usual lisp technique. We simply provide a fixnum array and use numeric indices to index into that array—thinking about it exactly like C. We then use declarations to get lisp to drop type and safety checks so it is compiled exactly like C. Finally, we use macros to make the whole process convenient and safe.

In general, indexing into an array is a complex, slow procedure. The compiler needs to check that your index is numeric, you are trying to index an array, and the index is within the bounds of the array. Furthermore, arrays of different types can can have different code to access the elements. With this book's code loaded, try evaluating the following form (`dis` is described in detail in

section 7.3, Getting To Know Your Disassembler):

```
(dis (arr ind)
  (aref arr ind))
```

Because `aref` can mean so many possible things when no types are known, your compiler will probably not inline the array access code. In the above disassembly output, you should see a function call to something like CMUCL's `data-vector-ref`. Exercise: Get the source code for your lisp environment and examine this function. In CMUCL it is in the file `array.lisp`. Also examine the other functions in that file, including `data-vector-set`. If your lisp environment doesn't come with complete source code, or you aren't able to do anything you want with the source code you do have, upgrade your COMMON LISP environment as soon as possible.

Just like COMMON LISP can inline the function + when it has enough type information, it can also inline `aref`. Try the following form:

```
(dis (((simple-array fixnum) arr)
      (fixnum ind))
  (aref arr ind))
```

The above should have removed the indirection to the general array reference function. Simple arrays are arrays of one dimension where the elements are adjacent in memory like C-style memory. In the above we specified `fixnum` as our array element, but your COMMON LISP environment probably also provides types for fixnums of different size, bytes, unsigned bytes, floats, double floats, and much more. Although the above doesn't contain the indirection, it still has lots of code that implements the type and safety checks we usually rely on when programming lisp. However, just as we can use the sharp-f read macro from *section 7.2, Macros Make Lisp Fast* to tell lisp to make arithmetic fast, the same can be done for array references:

```
(dis (((simple-array fixnum) arr)
```

```
    (fixnum ind))
#f
(aref arr ind))
```

Unlike our earlier **arefs**, the performance of this bit of code will not be dominated by type and safety checks. This is the code that should be used inside of performance critical loops. Notice that because we have removed almost all of the safety features from this code that it is just as dangerous as its C equivalent. In particular, it could suffer from *buffer overflow* problems[15]. With C you program this way everywhere. With lisp you program safely everywhere except where performance matters, tuning the *hot-spots* of the code to make your whole program run faster. Thanks to macros, these hot-spots can be arbitrarily small. There is no need to compile, for example, entire functions in fast/dangerous mode. Macros allow us to optimise narrow, specific parts of an expression. Fast code can transparently co-exist with safe code and macros let us trade away the least safety necessary in order to achieve the required performance.

Because if you have made it this far in the book you should already have a good grasp of macro authoring and declarations, there is not much more that needs to be said regarding pointer scoping. In short, C provides a very specific domain specific language for controlling the CPU based on fixnum arithmetic, but you can write much better languages using macros. Efficient pointer scoping (which we can now confess really means array access—closure examples notwithstanding) is mostly a matter of knowing how macros work, how declarations work, and how to read your disassembler.

An example macro that efficiently accesses arrays is **with-fast-stack**. This macro was chosen to give an opportunity to discuss something called *amortisation*. **With-fast-stack** implements a stack data-structure named by **sym**. Unlike COMMON LISP **push** and **pop** stacks that use cons cells to store stacks of elements of any type, these stacks use a simple array to store

[15] "Buffer overflow" is an umbrella term for a variety of possible security problems with C (and sometimes even lisp) programs.

Listing 7.12: WITH-FAST-STACK

```
(defmacro! with-fast-stack
           ((sym &key (type 'fixnum) (size 1000)
                      (safe-zone 100))
            &rest body)
  `(let ((,g!index ,safe-zone)
         (,g!mem (make-array ,(+ size (* 2 safe-zone))
                             :element-type ',type)))
     (declare (type (simple-array ,type) ,g!mem)
              (type fixnum ,g!index))
     (macrolet
       ((,(symb 'fast-push- sym) (val)
            `(locally #f
               (setf (aref ,',g!mem ,',g!index) ,val)
               (incf ,',g!index)))
        (,(symb 'fast-pop- sym) ()
            `(locally #f
               (decf ,',g!index)
               (aref ,',g!mem ,',g!index)))
        (,(symb 'check-stack- sym) ()
            `(progn
               (if (<= ,',g!index ,,safe-zone)
                 (error "Stack underflow: ~a"
                        ',',sym))
               (if (<= ,,(- size safe-zone)
                       ,',g!index)
                 (error "Stack overflow: ~a"
                        ',',sym)))))
        ,@body)))
```

elements of a fixed type which can be specified by the `:type` keyword. The array is also of a fixed size but this size can be chosen through the `:size` keyword. The stack is accessed by using some `macrolet` defined local macros. If your stack's name is `input`, the macros bound will be `fast-push-input`, `fast-pop-input`, and `check-stacks-input`. Examine a compiled expansion with `dis`:

```
* (dis ((fixnum a))
    (with-fast-stack (input :size 2000)
      (loop for i from 1 to 1000000 do
        (fast-push-input a))))
```

The `fast-push-input` operation compiles into very tight (and very unsafe) machine code:

```
;;; [8] (FAST-PUSH-INPUT A)
MOV     ECX, [EBP-20]
MOV     EDX, [EBP-16]
MOV     EAX, [EBP-12]
MOV     [ECX+EDX+1], EAX
MOV     EAX, [EBP-16]
ADD     EAX, 4
MOV     [EBP-16], EAX
```

But the looping was compiled safely as usual, implementing error checking and indirection to arithmetic functions, even though it is inside a `with-fast-stack` macro.

```
;;; [7] (LOOP FOR I FROM 1...)
...
CALL    #x100001D0  ; #x100001D0: GENERIC-+
...
CALL    #x10000468  ; #x10000468: GENERIC->
```

Clearly this loop will not run as fast as it could. Its performance will be dominated by the looping overhead, not the stack operation. If we needed speed, we could declare `i` to be a fixnum and add speed declarations to the loop as we have seen before.

Safe code can co-exist with fast code. The code we just disassembled is, of course, extremely dangerous. It doesn't ever check the height of the stack to see if we overflowed or underflowed past our boundaries. That was something we were trying to avoid for efficiency's sake. The solution offered by `with-fast-stack` is inspired by the word `stacks` in the *forth* programming language. By using the `check-stacks-input` local macro, our code can verify that the stack is within bounds and throw an error otherwise. Because forth is designed to perform well on the most limited of hardware platforms, forth *amortises* the cost of performing the bounds check. Instead of performing it after every operation like lisp will do by default, it only does it every N operations. In forth, this word is often invoked only after evaluating a form in the REPL (we will have more to say about forth in *chapter 8, Lisp Moving Forth Moving Lisp*). So instead of checking the bounds every operation, we could do it every 10 operations, perhaps reducing the bounds checking cost by 90%[16]. When we do check the stack we know that, at worst, it will be 10 elements out of bounds. Or maybe there is some convenient, non-performance-critical place in your code that the check macro can go.

Another feature of `with-fast-stack` is that the arrays it creates have *safe zones*. That is, it allocates extra memory on either side of the stack as a *run-away lane* in case you screw up. It doesn't mean that running into these safe zones is a good idea (especially the underflow one) but it is better than running into memory you didn't allocate.

As mentioned, the code we just assembled is very dangerous and can write fixnums into memory that wasn't allocated to it. Don't ever do this. Exercise: Do this. Here is what happened for me:

```
* (compile nil
    '(lambda (a)
       (declare (type fixnum a))
       (with-fast-stack (input :size 2000)
```

[16]Though counting which operation we are currently at can imply overhead too.

```
          (loop for i from 1 to 1000000 do
            (fast-push-input a)))))

#<Function>
NIL
NIL
```

The dangerous code compiled just fine. Let's try running it:

```
* (funcall * 31337)

NIL
```

Well, it wasn't the disaster we were afraid of. Did anything bad happen?

```
* (compile nil '(lambda () t))

; Compilation unit aborted.
```

Hm that doesn't sound good.

```
* (gc)
Help! 12 nested errors.
KERNEL:*MAXIMUM-ERROR-DEPTH* exceeded.
** Closed the Terminal

NIL
```

That definitely doesn't sound good. Because lisp is a process running on unix, it is also possible to receive signals indicating you have written outside your allocated virtual memory (called a *segfault*). CMUCL handles these as recoverable conditions (though you should almost always reload your lisp image):

```
Error in function UNIX::SIGSEGV-HANDLER:
   Segmentation Violation at #x58AB5061.
   [Condition of type SIMPLE-ERROR]
```

In these states, the lisp image is said to be *hosed.* Programs that have the potential to become hosed like this are security disasters waiting to happen. The difference between C and lisp is that while C has this potential almost everywhere, lisp has it almost nowhere. If we ever need to take the risks of array based pointer scoping, lisp macros are the least obtrusive and safest means for doing so. Of course you almost never want to take these risks—stick to closures.

7.5 Tlists and Cons Pools

This section is about memory management, but it might not tell you what you want to hear. I was reluctant to even include it because I was afraid of perpetuating an incorrect myth about lisp, the mistaken notion that consing is slow. Sorry, but it just isn't true; consing is actually fast[GC-IS-FAST]. Algorithms that minimise indefinite extent storage are usually ideal, of course, but most algorithms can be more easily and directly written by consing. Do not be afraid to cons when you need memory. In fact, sometimes an excellent optimisation that can be made in lisp is to adapt an algorithm into a form where it can be implemented with cons cells so as to benefit from a tuned lisp garbage collector. Just as writing your own hash-table implementation is probably a bad idea, hacking up your own memory allocation routines is probably just as dumb. That said, this section explains some ways to do it. Surprise, we do it with macros.

Before we come back to memory allocation, we take a quick related detour. Although COMMON LISP is the tool of choice for the professional lisp programmer, many of the best introduction-to-lisp textbooks have been written about *Scheme.* The generally most revered is *Structure and Interpretation of Computer Programs*[SICP] by Hal Abelson, Jerry Sussman, and Julie Sussman. SICP[17] has been alternatively idolised or endured by freshmen students at MIT, where it was first introduced, for decades. Scheme's appeal to academia has been deep and pervasive. Most

[17]Pronounced sick-pea.

Listing 7.13: TLISTS

```
(declaim (inline make-tlist tlist-left
                 tlist-right tlist-empty-p))

(defun make-tlist () (cons nil nil))
(defun tlist-left (tl) (caar tl))
(defun tlist-right (tl) (cadr tl))
(defun tlist-empty-p (tl) (null (car tl)))
```

macro professionals start their lisp experience with Scheme—only when they are ready to start programming serious macros do they migrate to the macro hacker's language: COMMON LISP.

But when you migrate, you always take things with you. There is no escaping your past—your roots. If your roots lie with Scheme and you've read SICP, you probably remember *queues* (also see [USEFUL-LISP-ALGOS1-CHAPTER3]). An alternative description of them, the one we use here, is from from another good Scheme book, *Schematics of Computation*[SCHEMATICS] and is called the *tlist*. A tlist is a data-structure named after its inventor, an Interlisp hacker named Warren Teitelman. Although tlists are provided as Scheme code in *Schematics of Computation*, we present them here as a port to COMMON LISP.

As we can see in the constructor, `make-tlist`, a tlist is just a cons cell. But instead of using the car as the element and cdr as the next cons like a regular list, a tlist uses the car to point to the first cons in the real list, and the cdr to point to the last. If the car of a tlist is `nil`, the tlist is considered *empty*. Unlike regular lists, empty tlists are distinct (not `eq`). For tlists, the car of the cons cell acting as a tlist points to a cons cell that holds the *left* element of the tlist. The cdr points to a cons that holds the *right*.

The functions `tlist-left` and `tlist-right` return the left and right elements of the tlist without modifying the tlist. If the tlist is empty, these functions return `nil`. If you only use these functions you will not be able to store `nil` in your tlist. Luckily you can check to see if a tlist is empty before you use it with the `tlist-empty-p` predicate, and can thus store `nil`.

Listing 7.14: TLIST-ADD

```
(declaim (inline tlist-add-left
                 tlist-add-right))

(defun tlist-add-left (tl it)
  (let ((x (cons it (car tl))))
    (if (tlist-empty-p tl)
      (setf (cdr tl) x))
    (setf (car tl) x)))

(defun tlist-add-right (tl it)
  (let ((x (cons it nil)))
    (if (tlist-empty-p tl)
      (setf (car tl) x)
      (setf (cddr tl) x))
    (setf (cdr tl) x)))
```

Because doing it is so easy, we have decided to tell the compiler that all of these functions can be *inlined*[18]. This will allow the lisp compiler to generate more efficient expansions for the tlist functions. In some languages that don't provide much compiler control—like C—primitive macro systems are used to ensure that functions like our tlist utility are inlined. In lisp, where we control the compiler completely, there is no need to use macros for this purpose. The macros in this chapter are about much more than inlining.

We can add elements to the left of a tlist with the `tlist-add-left` function, and to the right with `tlist-add-right`. Because a pointer to the end of the list is maintained, adding elements to either end of a tlist is a *constant time* operation with respect to the length of the tlist. However, in general, addition to a tlist is not quite a constant time operation because of the memory allocation overhead imposed by consing. Its use of `cons` means tlist addition usually incurs the aggregated overhead of garbage collection.

Only removing an item from the left of a tlist is supported

[18] `Declaim` is sort of the global version of `declare`.

Listing 7.15: TLIST-REM-LEFT

```
(declaim (inline tlist-rem-left))

(defun tlist-rem-left (tl)
  (if (tlist-empty-p tl)
    (error "Remove from empty tlist")
    (let ((x (car tl)))
      (setf (car tl) (cdar tl))
      (if (tlist-empty-p tl)
        (setf (cdr tl) nil)) ;; For gc
      (car x))))
```

Listing 7.16: TLIST-UPDATE

```
(declaim (inline tlist-update))

(defun tlist-update (tl)
  (setf (cdr tl) (last (car tl))))
```

with the given functions. Because we only keep a pointer to the first and last elements of the tlist, the only way to find the second-to-last element is to traverse the entire list, starting with the left of the tlist.

A tlist is a queue abstraction built on top of cons cells that is especially useful because it is a *transparent* data structure. While some data structures that implement tlist functionality—like queues—only provide you with a limited interface to the data structure, tlists are specified directly as cons cells. Instead of inventing some API to hopefully meet everyone's needs, Teitelman decided to tie the specification of the tlist directly to the lisp cons cell. This design decision is what separates the tlist from other queue implementations. When programming with transparent specifications, instead of making special API functions to do things, the code *is* the API.

If we decide to access the car of a tlist and modify its contents, we need to make sure that the tlist remains consistent. Assuming after our manipulation the desired list is stored in the tlist's car,

Listing 7.17: COUNTING-CONS

```
(defvar number-of-conses 0)

(declaim (inline counting-cons))

(defun counting-cons (a b)
  (incf number-of-conses)
  (cons a b))
```

Listing 7.18: WITH-CONSES-COUNTED

```
(defmacro! with-conses-counted (&rest body)
  `(let ((,g!orig number-of-conses))
     ,@body
     (- number-of-conses ,g!orig)))
```

we can use `tlist-update` to set the cdr appropriately[19].

So the main benefit of a tlist is to emulate regular lisp lists as closely as possible, while enabling an operation that adds elements to the end in constant time. Because tlists use `cons` just like regular lists, they have exactly the same memory overheads.

COMMON LISP doesn't specify much functionality for monitoring or controlling memory allocation. So let's write some. First off, recall from *section 3.5, Unwanted Capture* that we are not allowed to re-define or re-bind a function specified by COMMON LISP. We can't intercept calls to `cons` directly so we instead use a *wrapper*. `Counting-cons` is identical to `cons` except that every time it is called it increments the variable `number-of-conses`.

`With-conses-counted` is our primary interface for examining the value of `number-of-conses`. Its expansion will record its initial value, perform the operations provided in the macro body, then return the number of times that `counting-cons` was invoked.

An unfortunate consequence of our strategy of renaming `cons`

[19]Often there is a more efficient way of storing the last cons element of the list into the cdr of a tlist. Doing so can avoid the linear-in-length-of-list `tlist-update` operation. Since the tlist specification is transparent, both ways are correct.

Listing 7.19: COUNTING-PUSH

```
(defmacro counting-push (obj stack)
  `(setq ,stack (counting-cons ,obj ,stack)))
```

to counting-cons is that any routines we want to examine for memory performance need to be re-written to use counting-cons as in counting-push. Here we can see that each time it is invoked, counting-push calls counting-cons exactly once:

```
* (let (stack)
    (with-conses-counted
      (loop for i from 1 to 100 do
        (counting-push nil stack)
        (pop stack))))

100
```

The pop operator above remove the elements and the cons cells used to store them from the stack. What happens to these cons cells? They become garbage. Normally lisp spews out this garbage everywhere and nobody cares because COMMON LISP environments have excellent recycling programs called *garbage collectors* that reclaim this storage. However, collecting garbage isn't free—somebody has to pay for the garbage to be picked up, transported to wherever it goes, and processed into something again fit for use. What if we could create mini-recycling programs on-site? For example, the above loop invoked counting-cons 100 times, generating 100 pieces of garbage that need collecting. However, a quick look at the code reveals that stack never has more than one item on it at a time. If we recycled this one cons cell so it is available for counting-push again, we could avoid calling counting-cons to get another cons cell. This concept is known as a *cons pool*. In addition to reducing the pressure on the garbage collector, cons pools can also help improve the *locality* of data structures that allocated memory often.

With-cons-pool is one way we can create cons pools. Notice

Listing 7.20: WITH-CONS-POOL

```
(defmacro with-cons-pool (&rest body)
  `(let ((cons-pool)
         (cons-pool-count 0)
         (cons-pool-limit 100))
     (declare (ignorable cons-pool
                         cons-pool-count
                         cons-pool-limit))
     ,@body))
```

Listing 7.21: CONS-POOL-CONS

```
(defmacro! cons-pool-cons (o!car o!cdr)
  `(if (= cons-pool-count 0)
     (counting-cons ,g!car ,g!cdr)
     (let ((,g!cell cons-pool))
       (decf cons-pool-count)
       (setf cons-pool (cdr cons-pool))
       (setf (car ,g!cell) ,g!car
             (cdr ,g!cell) ,g!cdr)
       ,g!cell)))
```

that this macro expands into a let form, creating bindings for `cons-pool`, `cons-pool-count`, and `cons-pool-limit`. These variables will be used to store cons cells that can be recycled. Because it introduces variables invisibly, `with-cons-pool` is an *anaphoric macro*. Notice also that because COMMON LISP provides a *dual syntax* for lexical and dynamic variables that the anaphoric bindings this macro's expansion creates may be dynamic or lexical, depending on whether the anaphora are declared special at the site of the macro use or not.

`Cons-pool-cons` expands into some code that allocates cons cells from a cons pool. It assumes that it is inside the lexical scope of a `with-cons-pool`, or, if the anaphora are declared special, then there currently exists dynamic bindings for them. `Cons-pool-cons` only invokes `counting-cons` when its pool is empty. It will never store more than `cons-pool-limit` in the pool.

If we have determined that we no longer need a cons cell, we

Listing 7.22: CONS-POOL-FREE

```
(defmacro! cons-pool-free (o!cell)
  `(when (<= cons-pool-count
             (- cons-pool-limit 1))
     (incf cons-pool-count)
     (setf (car ,g!cell) nil)
     (push ,g!cell cons-pool)))
```

Listing 7.23: MAKE-CONS-POOL-STACK

```
(defmacro make-cons-pool-stack ()
  `(let (stack)
     (dlambda
       (:push (elem)
         (setf stack
               (cons-pool-cons elem stack)))
       (:pop ()
         (if (null stack)
           (error "Tried to pop an empty stack"))
         (let ((cell stack)
               (elem (car stack)))
           (setf stack (cdr stack))
           (cons-pool-free cell)
           elem)))))
```

can move it into the cons pool by freeing it with `cons-pool-free`. When this is done, the code must promise to never again access the cons cell that it just freed. The code that `cons-pool-free` expands into will push the freed cons cell onto `cons-pool` and increment `cons-pool-count` unless `cons-pool-count` is greater than `cons-pool-limit` in which case the cell will be left for garbage collection to collect. Notice that you are not required to `cons-pool-free` your cons cells when you have determined you no longer need them because the garbage collector will still be able to determine when they are no longer needed. Freeing them is simply an efficiency optimisation we can make if we know a little bit of extra information that lisp doesn't.

So the design of cons pools consists of two macros, one that

creates anaphora, invisibly introducing either lexical or special bindings, and another that invisibly consumes these anaphora. Typically, another macro is used to *combine* these macros. `Make-cons-pool-stack` is one such example. It creates a data structure similar to the COMMON LISP stack, which, of course, is really just a list updated with the **push** and pop macros. However, our data structure is different from **push** and pop because it isn't transparently specified. The implementation details of these stacks are separate from how they are actually used. This is important because we do not want to require users of our stacks to use their own methods to push and pop data, instead we want them to use our memory-optimised versions. `Make-cons-pool-stack` use `dlambda` from *section 5.7, Dlambda.* Here is an example where we create a lexical cons pool enclosing a new stack data structure, and then push and pop an item 100 times:

```
* (with-cons-pool
    (let ((stack (make-cons-pool-stack)))
      (with-conses-counted
        (loop for i from 1 to 100 do
          (funcall stack :push nil)
          (funcall stack :pop)))))

1
```

Notice that `counting-cons`—which is the only memory allocation function used—is only invoked once. The one cons cell that is ever required is recycled instead of being collected. If this loop occurred in compiled code, and the loop iterated enough times, the cons pool version can be expected to execute faster, simply because the garbage collector will not be invoked. Often more importantly, our loop will not have unexpected pauses in execution when the garbage collector runs. Of course, we almost never notice these pauses because lisp is smart enough to not do full garbage collections at once, instead *amortising* the operation with a technique known as *incremental collection.* Garbage collectors also implement an optimisation called *generational collection* where recently allocated memory is collected more often than old

Listing 7.24: MAKE-SHARED-CONS-POOL-STACK

```
(with-cons-pool
  (defun make-shared-cons-pool-stack ()
    (make-cons-pool-stack)))
```

Listing 7.25: WITH-DYNAMIC-CONS-POOLS

```
(defmacro with-dynamic-cons-pools (&rest body)
  `(locally (declare (special cons-pool
                              cons-pool-count
                              cons-pool-limit))
     ,@body))
```

memory. Surprisingly, this turns out to be a type of referencing counting[UNIFIED-THEORY-OF-GC].

But with cons pools we can cons less (or not at all) and thus reduce (or eliminate) garbage collection execution time indeterminism. Most lisp systems also have a way to temporarily disable the garbage collector so you can execute something without pausing and, instead, pause a bit longer at some point in time where you don't care about such pauses. In CMUCL you can use the `gc-on` and `gc-off` functions. Also see the code in `signal.lisp`. Exercise: Disable the garbage collector then cons up a bunch of garbage in a loop. Use the unix `top` program to monitor your memory usage.

Although the above stack implementation requires you to preserve locality in the same lexical context with a `with-cons-pool` to indicate the stacks you want to share a cons pool, thanks to the transparent design of these macros, we can combine them with closures to indicate this locality however we like. `Make-shared-cons-pool-stack` works the same as `make-cons-pool-stack` except that it doesn't require you wrap `with-cons-pool` around them. These variables have already been captured. So all stacks created with `make-shared-cons-pool-stack` will share the same cons pool.

Thanks to the duality of syntax between lexical and special variables, we can choose to use the dynamic environment to store

Listing 7.26: FILL-CONS-POOL

```
(defmacro fill-cons-pool ()
  `(let (tp)
     (loop for i from cons-pool-count
                  to cons-pool-limit
           do (push
                (cons-pool-cons nil nil)
                tp))
     (loop while tp
           do (cons-pool-free (pop tp)))))
```

our cons pools. The `with-dynamic-cons-pools` macro makes any cons pool references in its lexical scope refer to the dynamic bindings of the anaphora. One strategy is to wrap all code that uses cons pools with `with-dynamic-cons-pools` then, when you actually execute your program, have dynamic bindings created for the cons pool. Because you can shadow dynamic bindings with new dynamic bindings, you can preserve locality to any dynamic granularity. To create dynamic bindings, simply wrap `with-dynamic-cons-pools` around a `with-cons-pool`.

Especially when trying to reduce garbage collection execution time indeterminism, it can be necessary to ensure a cons pool has available cells in its pool so that the program will not cons at all (assuming we don't exhaust the pool). To do this, simply cons the required cells initially—when it is acceptable to cons-then add them to the pool with `fill-cons-pool`, filling the cons pool up to its `cons-pool-limit`.

Memory is a very complicated topic and its efficiency implications depend on your hardware, your lisp implementation, and inevitable advances in technology. Unless you really know what you're doing, trying to improve on your system's memory routines will probably be more trouble than it is worth. Systems programmers have been tweaking memory for as long as there has been systems. They will certainly be doing it for a while yet. Memory management is hard—the only thing that is certain is that macros are the best tool for doing it.

7.6 Sorting Networks

There is no better tool for experimenting with efficiency or actually implementing efficient programs than lisp. Lisp is unique because not only does it allow us to concentrate on smart algorithms and designs, but also lets us exploit these algorithms and designs to their maximal efficiency potential using state of the art machine-code compilers. This section describes, from a lisp perspective, a corner of computer science that has been extensively studied but still nowhere near exhausted: sorting. Most people consider sorting a solved problem so you may surprised to learn that there are still many important open questions.

We know of many excellent general-purpose sorting algorithms. Algorithms like *quick sort* are the most common because they efficiently sort large quantities of data. But if, instead, we wish to sort many small batches of data, general-purpose sorting algorithms like quick sort can be overkill. This section is about a solution to this problem that many people have been obsessed with for decades but which is still very fertile ground for research. Most importantly to us, this solution gives an opportunity to show advanced optimisation techniques that are straightforward in lisp but are such major undertakings in most other languages that they are hardly worth it. In this section and the next, we will re-implement a macro described by Graham in *On Lisp* called `sortf`[ON-LISP-P176]. While Graham's `sortf` was designed to illustrate how to write macros using generalised variables, ours is designed for speed. For certain circumstances, our `sortf` will achieve order-of-magnitude improvements over our system's well-tuned `sort` function.

This section is dedicated to my teacher and friend Alan Paeth who taught me, among many other things, that even sorting can be interesting. I also gratefully acknowledge John Gamble and his excellent Perl program, `Algorithm-Networksort`[ALGORITHM-NETWORKSORT]. This program was used to experiment with the different algorithms and also to generate the ASCII art networks that appear in this section.

A sorting network is an algorithm for *obliviously* sorting data-

sets of a particular fixed size. That is, unlike most algorithms like quick sort, the operation of a sorting network does not depend on the particular data-set it is used to sort. Every step of the sort was decided when the network was designed. A sorting network is a simple list of pairs of indices into a data-set. Each of these pairs corresponds to the indices that should be used in a compare-swap operation. After performing all of these compare-swap operations in sequence, the elements will be in sorted order.

Algorithms like quick sort that are excellent for large data-sets can have unacceptable overheads for certain classes of sorting problems. First, quick sort implementations usually allow you to choose a custom comparison operator so as to make the sorting code more general. This means that every comparison will entail a function call to the comparison function instead of, say, being implemented as inline machine code. Second, because quick sort implementations are so general, they often can't take advantage of optimisations that can be made when we know our data-sets are of particular small, fixed sizes. Third, often we don't want to completely sort a data-set, but instead sort only enough of it to determine a certain element—perhaps the median element. Sorting networks that don't bother to find the complete ordering are sometimes called *selection networks.*

To clarify the notion of a sorting network, as well as to give an idea of how subtle and counterintuitive the topic can be, we consider some of the simplest possible networks: ones that sort three elements. Most programmers know that sorting three elements can be done easily with three comparisons and often don't bother using quick sort when there are exactly three elements. It is easy to convince yourself that these compare-swap operations can be executed in any order and the result will be the same. But it is not nearly so obvious that some of the orderings are inherently less efficient than others.

The network `bad-3-sn` might be the most obvious implementation of a three element network but is—as the name suggests—not the optimal one. The ASCII art picture helps to visualise the algorithm described by the list-based network description in `bad-3-sn`. The algorithm says to compare the elements at indices 0 and 1 of

Listing 7.27: BAD-3-SN

```
(defvar bad-3-sn
  '((0 1) (0 2) (1 2)))
```

Listing 7.28: BAD-3-SN-PIC

```
o--^--^-----o
   |  |
o--v--|--^--o
      |  |
o-----v--v--o
```

a data-set and, if they are out of order, swap them into correct order. Perform the same operation for the pair of indices (0 2) and then finally for (1 2). After this process, the elements will be sorted. If we implemented this sorting network as code to sort an array of length three, call it a, then it might look like this[20]:

```
(progn
  (if (> (aref a 0) (aref a 1))
    (rotatef (aref a 0) (aref a 1)))
  (if (> (aref a 0) (aref a 2))
    (rotatef (aref a 0) (aref a 2)))
  (if (> (aref a 1) (aref a 2))
    (rotatef (aref a 1) (aref a 2))))
```

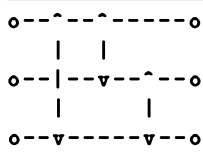
Bad-3-sn is correct but inefficient compared to good-3-sn. By exchanging the order of the first two comparison-swap operations, we achieve a more efficient network. On average, this network will

[20] Rotatef is a generic lisp swapping operator.

Listing 7.29: GOOD-3-SN

```
(defvar good-3-sn
  '((0 2) (0 1) (1 2)))
```

Listing 7.30: GOOD-3-SN-PIC



Listing 7.31: INTERPRET-SN

```
(defvar tracing-interpret-sn nil)

(defun interpret-sn (data sn)
  (let ((step 0) (swaps 0))
    (dolist (i sn)
      (if tracing-interpret-sn
        (format t "Step ~a: ~a~%" step data))
      (if (> #1=(nth (car i) data)
             #2=(nth (cadr i) data))
        (progn
          (rotatef #1# #2#)
          (incf swaps)))
      (incf step))
    (values swaps data)))
```

perform fewer swap operations than will bad-3-sn. The best way to describe this is with *conditional probability* but because this is a book about lisp, and not sorting networks, we will shy away from this. Instead, we show that good-3-sn is better than bad-3-sn by enumerating all the permutations and then measuring the number of swaps that take place when we interpret them with the two networks. For now here is an intuitive explanation: if the long link in the network is performed first, then after this first operation at least one of the minimum or maximum elements will be in its correct, final position. So at least one of the subsequent comparison-swap operations will not perform a swap. If, however, a short link is performed first then it is possible that neither of these elements will be in their final positions and will both require future swapping.

To explore this phenomenon, we implement an interpreter for sorting networks, **interpret-sn**. This interpreter will apply a sorting network **sn** to a data-set represented by a list. It will return the number of swaps that were performed as the first value and the resulting sorted data-set as the second. Notice the use of the **#=** and **##** self-referential read macros used to avoid re-typing the accessor forms. Notice also the use of a tracing variable that we can bind to a non-null value if we want to see the step-by-step sorting process. To start, consider a data-set that is already sorted. Obviously both **bad-3-sn** and **good-3-sn** perform no swapping:

```
* (let ((tracing-interpret-sn t))
    (interpret-sn '(1 2 3) bad-3-sn))
Step 0: (1 2 3)
Step 1: (1 2 3)
Step 2: (1 2 3)
0
(1 2 3)
* (let ((tracing-interpret-sn t))
    (interpret-sn '(1 2 3) good-3-sn))
Step 0: (1 2 3)
Step 1: (1 2 3)
Step 2: (1 2 3)
0
(1 2 3)
```

Next, consider a case where every element is out of sequence. Again, both sorting networks perform identically, performing the necessary two swaps:

```
* (let ((tracing-interpret-sn t))
    (interpret-sn '(3 1 2) bad-3-sn))
Step 0: (3 1 2)
Step 1: (1 3 2)
Step 2: (1 3 2)
2
(1 2 3)
```

```
* (let ((tracing-interpret-sn t))
    (interpret-sn '(3 1 2) good-3-sn))
Step 0: (3 1 2)
Step 1: (2 1 3)
Step 2: (1 2 3)
2
(1 2 3)
```

However, here is a case where bad-3-sn results in a *worst-case* behaviour of three swaps:

```
* (let ((tracing-interpret-sn t))
    (interpret-sn '(3 2 1) bad-3-sn))
Step 0: (3 2 1)
Step 1: (2 3 1)
Step 2: (1 3 2)
3
(1 2 3)
* (let ((tracing-interpret-sn t))
    (interpret-sn '(3 2 1) good-3-sn))
Step 0: (3 2 1)
Step 1: (1 2 3)
Step 2: (1 2 3)
1
(1 2 3)
```

In the above, bad-3-sn performed three swaps where the optimal good-3-sn performed only one. Shouldn't there be a symmetric case where good-3-sn performs poorly? It turns out no, good-3-sn is really just better. If you still don't believe this, investigate the *Monty Hall problem* to get a feel for just how counterintuitive these sorts of problems can be. So it seems the moral is to always swap elements into their correct positions as soon as possible so that the least amount of swaps will take place.

To quantify how much better good-3-sn is than bad-3-sn we present a utility all-sn-perms that generates all permutations of the numbers from 1 to n. All-sn-perms embodies many lisp patterns, including recursively consing up a network of connected,

Listing 7.32: ALL-SN-PERMS

```
(defun all-sn-perms (n)
  (let (perms curr)
    (funcall
      (alambda (left)
        (if left
          (loop for i from 0 to (1- (length left)) do
            (push (nth i left) curr)
            (self (append (subseq left 0 i)
                          (subseq left (1+ i))))
            (pop curr))
          (push curr perms)))
      (loop for i from 1 to n collect i))
    perms))
```

temporary lists, and the use of Graham's anaphoric `alambda` macro. Here, we generate all 6 (factorial of 3) permutations of the numbers 1 to 3:

```
* (all-sn-perms 3)

((1 2 3) (2 1 3) (1 3 2)
 (3 1 2) (2 3 1) (3 2 1))
```

Notice that because of how `all-sn-perms` is written, the above lists share structure with one another so when using them for interpreting sorting networks (a destructive operation) we should always make sure to sort copies of them, as in `average-swaps-calc`. For problems with results that can be structured this way, sharing structure like this is generally a good programming technique because it can reduce the total memory required for your data-structures[21].

Using our `interpret-sn` sorting network interpreter, we can use the actual numbers of swaps that it records for each possible permutation with `average-swaps-calc`. This function simply loops across each permutation, applying the interpreter against the given sorting network, summing the swaps that occurred, and

[21] Though here it was only because it seemed the simplest way to code it.

Listing 7.33: AVERAGE-SWAPS-CALC

```
(defun average-swaps-calc (n sn)
  (/ (loop for i in (all-sn-perms n) sum
       (interpret-sn (copy-list i) sn))
     (fact n)))
```

then returning this sum divided by the number of possible permutations. If we make the assumption that every possible permutation is equally likely, this calculation represents the average number of swaps that will take place for every sort. Here we see that bad-3-sn results in, on average, 1.5 swaps per sort:

```
* (average-swaps-calc 3 bad-3-sn)

3/2
```

Where good-3-sn results in, on average, only 1.166... swaps:

```
* (average-swaps-calc 3 good-3-sn)

7/6
```

Up until now, our sorting networks have been only able to sort data-sets of size three. Are there algorithms for generating sorting networks of arbitrary size? Yes, and these algorithms have been known for some time. In 1968, Ken Batcher described his ingenious algorithm[SN-APPLICATIONS] named by Donald Knuth as *merge exchange sort* or *algorithm 5.2.2M* from [TAOCP-VOL3-P111]. Batcher's algorithm is kind of a combination of *shell sort* and *merge sort* except that given a known input size, the comparison-swap operations it will make are completely determined independent of the data itself—exactly what we need for sorting networks. So to create a sorting network we run Batcher's algorithm and record what comparison-swap operations were made. Later we can inline these operations into a function for this particular input size. This process is not completely unlike *loop unrolling*, except that lisp allows us to take it much further.

Listing 7.34: BUILD-BATCHER-SN

```
(defun build-batcher-sn (n)
  (let* (network
         (tee (ceiling (log n 2)))
         (p (ash 1 (- tee 1))))
    (loop while (> p 0) do
      (let ((q (ash 1 (- tee 1)))
            (r 0)
            (d p))
        (loop while (> d 0) do
          (loop for i from 0 to (- n d 1) do
            (if (= (logand i p) r)
              (push (list i (+ i d))
                    network)))
          (setf d (- q p)
                q (ash q -1)
                r p)))
      (setf p (ash p -1)))
    (nreverse network)))
```

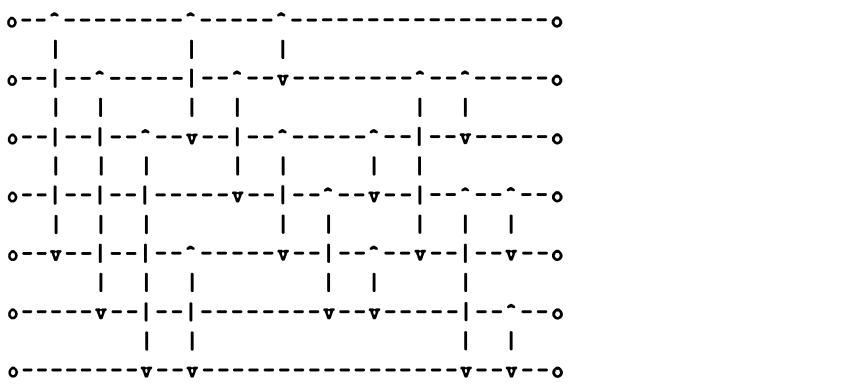
Build-batcher-sn is a lisp implementation of Batcher's algorithm directly transcribed from Knuth's description. Thanks to lisp's arbitrary precision support for bit-wise integer operations, this implementation doesn't suffer any artificial size limitations at **n** of, say, 32 or 64. We can use **build-batcher-sn** to easily construct efficient sorting networks of any size. Here is its construction for a network of size three—the same as **good-3-sn** above:

```
* (build-batcher-sn 3)

((0 2) (0 1) (1 2))
```

And here is its construction for a network of size seven:

```
* (build-batcher-sn 7)

((0 4) (1 5) (2 6) (0 2) (1 3) (4 6) (2 4)
 (3 5) (0 1) (2 3) (4 5) (1 4) (3 6) (1 2)
 (3 4) (5 6))
```

Listing 7.35: BATCHER-7-SN-PIC



Batcher's networks are good but are known to be slightly sub-optimal for most network sizes. While better networks for many particular sizes have been discovered, how to find these better networks, as well as whether they are optimal or not, is an important unsolved problem. This area of research has had important advances made by *evolutionary algorithms* that use novel artificial intelligence techniques to effectively search the super-exponential spaces of sorting network problems. For instance, the best network of size thirteen currently known was discovered by the *Evolving Non-Determinism* algorithm[END].

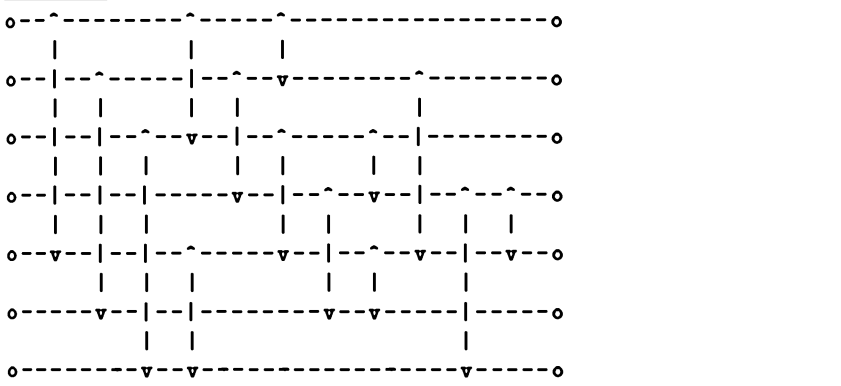
The ASCII art representations of sorting networks shown here were created by John Gamble's excellent `Algorithm-Networksort` Perl program. Notice that the picture puts some links that can be performed in parallel together into the same vertical column. This shows that sorting networks are algorithms that can, at least in specialised hardware, benefit from parallelism in the comparison-swap operations. Discovering how to create good parallel sorting networks, along with just how parallel we can make them, also remain important, mostly open problems.

Above we mentioned one disadvantage of general-purpose sorting functions is that they are hard-coded into performing the entire sort operation. If we like, we can sort the data-set just

Listing 7.36: PRUNE-SN-FOR-MEDIAN

```
(defun prune-sn-for-median (elems network)
  (let ((mid (floor elems 2)))
    (nreverse
      (if (evenp elems)
        (prune-sn-for-median-aux
          (reverse network)
          (list (1- mid) mid))
        (prune-sn-for-median-aux
          (reverse network)
          (list mid))))))

(defun prune-sn-for-median-aux (network contam)
  (if network
    (if (intersection (car network) contam)
      (cons (car network)
            (prune-sn-for-median-aux
              (cdr network)
              (remove-duplicates
                (append (car network) contam))))
      (prune-sn-for-median-aux
        (cdr network) contam))))
```

Listing 7.37: HOYTE-7-MEDIAN-SN-PIC



enough to know for certain that one element is in its final position. Typically, the element we are interested in is the middle, or *median* element. The functions **prune-sn-for-median** and **prune-sn-for-median-aux** employ a modest, mostly obvious algorithm I have discovered that can eliminate many of the unnecessary comparison-swap operations so as to construct arbitrary selection networks.

The algorithm starts with a Batcher network then works backwards, keeping track of *contaminated* elements—elements that can't have any of their existing links removed because doing so will change the outcome of the network for that element. Any links that connect uncontaminated elements can be removed without changing the outcome for the contaminated elements. Every link that connects a contaminated element to an uncontaminated link contaminates the uncontaminated element. When we contaminate just the middle element (or the two middle elements in case of even input sizes) we create a median selection network.

The algorithm's output for size 7, a modified Batcher network with two of its links removed, is shown. After running this network, the median element will be in the correct position, but other elements are not guaranteed to be sorted. As an example, here we only sort the list enough to discover that 4 is the median element:

Listing 7.38: PRUNE-SN-FOR-MEDIAN-CALC

```
(defun prune-sn-for-median-calc (n)
  (loop for i from 2 to n collect
    (let* ((sn (build-batcher-sn i))
           (snp (prune-sn-for-median i sn)))
      (list i
           (length sn)
           (length snp)))))
```

```
* (interpret-sn
    '(4 2 3 7 6 1 5)
    (prune-sn-for-median
      7 (build-batcher-sn 7)))

6
(1 3 2 4 5 7 6)
```

For networks of size seven, our modified median Batcher networks perform 12 comparison-swap operations versus the 14 operations that are performed by the regular Batcher network. `Prune-sn-for-median-calc` gives us the data on this class of networks for different size sorting networks. It computes the Batcher network for sizes up to `n` and groups their size with the size of the associated median network created by our algorithm.

Network sizes up to 49 are calculated. Notice that at the smallest sizes, very few, if any, operations are saved. But for slightly larger numbers, we begin to save roughly 20% of the comparison-swap operations from being performed. When we are only concerned with medians, these networks are good choices. However, the construction of optimal median sorting networks is also an open area of research. The modified Batcher networks developed in this chapter are decent but still far from optimal. The best median selection networks for sizes 9 and 25 (3x3 and 5x5 image kernel sizes) currently known were discovered by Paeth[GRAPHICS-GEMS-P171-175] and are presented here and included with this book's code. Here are the lengths for Paeth's median networks:

Listing 7.39: PRUNED-MEDIAN-DATA

```
* (prune-sn-for-median-calc 49)

((2 1 1) (3 3 3) (4 5 5) (5 9 8) (6 12 12)
 (7 16 14) (8 19 17) (9 26 22) (10 31 29)
 (11 37 31) (12 41 35) (13 48 40) (14 53 47)
 (15 59 49) (16 63 53) (17 74 61) (18 82 72)
 (19 91 75) (20 97 81) (21 107 88) (22 114 98)
 (23 122 100) (24 127 105) (25 138 113) (26 146 124)
 (27 155 127) (28 161 133) (29 171 140) (30 178 150)
 (31 186 152) (32 191 157) (33 207 169) (34 219 185)
 (35 232 190) (36 241 199) (37 255 209) (38 265 223)
 (39 276 226) (40 283 233) (41 298 244) (42 309 259)
 (43 321 263) (44 329 271) (45 342 280) (46 351 293)
 (47 361 295) (48 367 301) (49 383 313))
```

Listing 7.40: PAETH-9-MEDIAN-SN

```
(defvar paeth-9-median-sn
  '((0 3) (1 4) (2 5) (0 1) (0 2) (4 5) (3 5) (1 2)
    (3 4) (1 3) (1 6) (4 6) (2 6) (2 3) (4 7) (2 4)
    (3 7) (4 8) (3 8) (3 4)))
```

Listing 7.41: PAETH-25-MEDIAN-SN

```
(defvar paeth-25-median-sn
  '((0 1) (3 4) (2 4) (2 3) (6 7) (5 7) (5 6) (9 10)
    (8 10) (8 9) (12 13) (11 13) (11 12) (15 16)
    (14 16) (14 15) (18 19) (17 19) (17 18) (21 22)
    (20 22) (20 21) (23 24) (2 5) (3 6) (0 6) (0 3)
    (4 7) (1 7) (1 4) (11 14) (8 14) (8 11) (12 15)
    (9 15) (9 12) (13 16) (10 16) (10 13) (20 23)
    (17 23) (17 20) (21 24) (18 24) (18 21) (19 22)
    (8 17) (9 18) (0 18) (0 9) (10 19) (1 19) (1 10)
    (11 20) (2 20) (2 11) (12 21) (3 21) (3 12)
    (13 22) (4 22) (4 13) (14 23) (5 23) (5 14)
    (15 24) (6 24) (6 15) (7 16) (7 19) (13 21)
    (15 23) (7 13) (7 15) (1 9) (3 11) (5 17) (11 17)
    (9 17) (4 10) (6 12) (7 14) (4 6) (4 7) (12 14)
    (10 14) (6 7) (10 12) (6 10) (6 17) (12 17)
    (7 17) (7 10) (12 18) (7 12) (10 18) (12 20)
    (10 20) (10 12)))
```

```
* (length paeth-9-median-sn)

20
* (length paeth-25-median-sn)

99
```

For networks of size 9, Batcher's full sorting network performs 26 operations. The best currently known was discovered by Floyd and is 25 operations. Our pruned median version of Batcher's network performs 22, and Paeth's median network 20. For networks of size 25, Batcher: 138, pruned: 113, Paeth: 99. So our median networks seem about 10% away from Paeth's, the currently best known median networks for these sizes. As expected, we can't prune any extra operations off Paeth's networks:

```
* (length (prune-sn-for-median
            9 paeth-9-median-sn))

20
* (length (prune-sn-for-median
            25 paeth-25-median-sn))

99
```

In theory, this is all very interesting. But in practice, theory is pretty boring. We developed all these list-based sorting networks, some that perform full sorts using Batcher's algorithm, and some that employ a contamination optimisation to Batcher's algorithm to find medians. Then we developed a toy interpreter for these networks that will no doubt perform horribly when compared to real sorting routines. What does any of this have to do with efficiency? Were our experiments just generating theoretical results instead of useful code[22]? In most languages the results of these experiments—our sorting networks—would be represented

[22]There is nothing necessarily wrong with theoretical results. Some of the world's most important inventions were developed for the sake of theory—arguably even lisp itself.

in some high-level data structure and not really be good for much. But in lisp these networks are already extremely efficient sorting programs; we just haven't written a compiler for them yet.

Exercise: Adapt the pruning algorithm (and its contamination approach) so it produces quartile selection networks. These are networks where not only the median is determined but also the median elements of the high and low halves of ordered elements.

7.7 Writing and Benchmarking Compilers

A *compiler* is a scary concept to most programmers because most languages are bad for writing them. Here is an analogy: parsing a complicated log file might be an intimidating, error-prone prospect to a programmer who only knows C or assembly, but thanks to Perl and regular expressions it is a non-issue to us multilingual programmers. Similarly, designing a powerful, expressive programming language and then creating a compiler to convert programs in this language into efficient machine code would be an intimidating task if we didn't know lisp. Lisp's advantage when it comes to writing compilers doesn't just make it a little bit better than other languages—it actually makes a new level of expression. In general this advantage is the difference between being able to do something and not being able to do something. Lisp programmers use compilers everywhere, and in ways and for tasks that non-lisp programmers sometimes don't even believe. How many C programmers have considered the interpretation overhead of the `printf` function described (and overcome) in *section 7.2, Macros Make Lisp Fast*? How many would try writing a compiler for `printf`? In lisp this is par for the course. Everything should compile down to lisp.

What is a compiler? If you're coming from Blub, the answer is probably buried in a large stack of books explaining parsing, syntax directed translation, context free grammars, etc. But don't worry, this is lisp and compilers are easy. So easy that if you have ever done any amount of serious lisp programming you have already written them, perhaps without even realising it. Another name for a compiler is a "macro". A macro compiles programs

Listing 7.42: SN-TO-LAMBDA-FORM-1

```
(defun sn-to-lambda-form% (sn)
  `(lambda (arr)
     #f
     (declare (type (simple-array fixnum) arr))
     ,@(mapcar
         #`(if (> #1=(aref arr ,(car a1))
                  #2=(aref arr ,(cadr a1)))
               (rotatef #1# #2#))
         sn)
     arr))
```

from one language to another. Lisp is actually all about writing these compilers—everything else is secondary. In lisp the only non-trivial aspect of compiler design is how to preserve the correctness of the target program while at the same time discovering efficient expansions for it. In other words, the essence of your compilation problem. So far, we've seen how to use macros to create custom languages fitted exactly to the task at hand, and also how to make lisp code efficient by using declarations to remove dualities and safety checks. Effective compiler writing is merely combining these two skills.

When we were creating a compiler macro to handle `format` in *section 7.2, Macros Make Lisp Fast*, what did the `formatter` compiler expand into? It was a lambda form[23]. Compiling to lambda forms sometimes makes sense because we can then use the `compile` function to convert them to machine code directly. Returning to our sorting networks from the previous chapter, `sn-to-lambda-form%` is a function that returns a lambda form. This lambda form will have an instruction for every comparison-swap operation in a list-based sorting network. Each instruction will (unsafely) index into a fixnum array, comparing and possibly using `rotatef` to swap elements. The fixnum array will be passed as an argument (`arr`) to the function created by this lambda form. That's about all there is to a decent machine code compiler. As

[23]Sharp-quoted lambda form actually.

with all lambda forms, thanks to the `lambda` macro we are able to evaluate them to get functions:

```
* (eval
    (sn-to-lambda-form%
      (build-batcher-sn 3)))

#<Interpreted Function>
```

Which can become compiled functions simply by calling `compile` on them:

```
* (compile nil *)

#<Function>
```

Let's look at the `disassemble` output (the compiled expansion):

```
* (disassemble *)
...
;;; (> (AREF ARR 0) (AREF ARR 2))
      9E:       MOV     EAX, [EDX+1]
      A1:       MOV     ECX, [EDX+9]
      A4:       CMP     EAX, ECX
      A6:       JLE     L0
;;; (ROTATEF (AREF ARR 0) (AREF ARR 2))
      A8:       MOV     EAX, [EDX+9]
      AB:       MOV     ECX, [EDX+1]
      AE:       MOV     [EDX+1], EAX
      B1:       MOV     [EDX+9], ECX
;;; (> (AREF ARR 0) (AREF ARR 1))
      B4: L0:   MOV     EAX, [EDX+1]
      B7:       MOV     ECX, [EDX+5]
      BA:       CMP     EAX, ECX
      BC:       JLE     L1
;;; (ROTATEF (AREF ARR 0) (AREF ARR 1))
      BE:       MOV     EAX, [EDX+5]
```

```
      C1:       MOV     ECX, [EDX+1]
      C4:       MOV     [EDX+1], EAX
      C7:       MOV     [EDX+5], ECX
;;; (> (AREF ARR 1) (AREF ARR 2))
      CA: L1:   MOV     EAX, [EDX+5]
      CD:       MOV     ECX, [EDX+9]
      D0:       CMP     EAX, ECX
      D2:       JLE     L2
;;; (ROTATEF (AREF ARR 1) (AREF ARR 2))
      D4:       MOV     EAX, [EDX+9]
      D7:       MOV     ECX, [EDX+5]
      DA:       MOV     [EDX+5], EAX
      DD:       MOV     [EDX+9], ECX
      E0: L2:   ...
```

The above machine code is fast, but it could be faster. Lisp compilers are smart—some of the smartest around—but they could always still be smarter. In the rare instances where we care about performance, examining the compiled expansions is critical because it's hard to know how smart your lisp implementation is. In the above assembly, if we look carefully, we will see that it is actually performing an unnecessary read operation every time it performs a swap. The problem is that `rotatef` expands into a redundant access. A *sufficiently smart compiler* could have figured out that we already have this value in a register and could have avoided the array access. But mine didn't so I re-structured the code so it resulted in a more efficient expansion.

`Sn-to-lambda-form` is the improved version of `sn-to-lambda-form%`. It creates temporary bindings for the variables read in so the array read instruction isn't re-performed for the swap operation. Here is the superior compiled expansion:

```
* (disassemble
    (compile nil
      (sn-to-lambda-form%
        (build-batcher-sn 3))))
...
;;; (LET ((A (AREF ARR 0)) (B (AREF ARR 2))) ...)
```

Listing 7.43: SN-TO-LAMBDA-FORM

```
(defun sn-to-lambda-form (sn)
  `(lambda (arr)
     #f
     (declare (type (simple-array fixnum) arr))
     ,@(mapcar
         #`(let ((a #1=(aref arr ,(car a1)))
                 (b #2=(aref arr ,(cadr a1))))
             (if (> a b)
               (setf #1# b
                     #2# a)))
         sn)
     arr))
```

```
      2E:       MOV     EAX, [EDX+1]
      31:       MOV     ECX, [EDX+9]
      34:       CMP     EAX, ECX
      36:       JLE     L0
;;; (SETF (AREF ARR 0) B (AREF ARR 2) A)
      38:       MOV     [EDX+1], ECX
      3B:       MOV     [EDX+9], EAX
;;; (LET ((A (AREF ARR 0)) (B (AREF ARR 1))) ...)
      3E: L0:   MOV     EAX, [EDX+1]
      41:       MOV     ECX, [EDX+5]
      44:       CMP     EAX, ECX
      46:       JLE     L1
;;; (SETF (AREF ARR 0) B (AREF ARR 1) A)
      48:       MOV     [EDX+1], ECX
      4B:       MOV     [EDX+5], EAX
;;; (LET ((A (AREF ARR 1)) (B (AREF ARR 2))) ...)
      4E: L1:   MOV     EAX, [EDX+5]
      51:       MOV     ECX, [EDX+9]
      54:       CMP     EAX, ECX
      56:       JLE     L2
;;; (SETF (AREF ARR 1) B (AREF ARR 2) A)
      58:       MOV     [EDX+5], ECX
      5B:       MOV     [EDX+9], EAX
```

```
5E: L2:    ...
```

Becoming familiar with your lisp compiler so that you know how efficient your macro expansions are is very important for writing efficient lisp. `Disassemble`, the source code to your lisp system, benchmarking tools like the `time` macro, and lots of patience is, unfortunately, the only way to really gain an intuition for how to write fast lisp code.

Expanding into a lambda form like our `sn-to-lambda-form` macro does might be the most obvious way to implement a compiler if you come from Blub languages. The source code to lambda form to `compile` to `disassemble` cycle feels very much like the edit, compile, disassemble cycle of Blub. You put source code in and get machine-code out. However, this approach could be more lispy. In lisp we generally create our compilers to be invisible—incorporated directly into other lisp programs. Ideally the `compile` function is never invoked until we want stuff to run fast. Macros shouldn't bother compiling all the way down to machine code, instead only enough to create a good expansion so that the compiler, whenever it is run, will have enough information available to make the complete program efficient.

We especially don't want to call `compile` at run-time. `Compile` is an expensive operation because of the many levels of macros that need to be expanded to compile something. Instead of calling `compile` at run-time, remember that lisp will have already compiled all the lambda forms inside a compiled function. Because of this ability to construct closures of already compiled code at run-time, it is easy to ensure that as much computation at compile-time is done as possible while still creating arbitrary functions (closures) at run-time.

`Sortf` is my favourite macro in this book. Not only is it concise, elegant, and an excellent demonstration of many of the macro techniques so far described, but it is also a useful piece of production code capable of performing extremely fast sorting operations. Best yet, this macro blends so nicely with lisp programs that it is effortless to use. We don't have to go far out of our way to benefit from this advanced lisp optimisation. Any time we need

Listing 7.44: SORTF

```
(defmacro! sortf (comparator &rest places)
  (if places
    `(tagbody
       ,@(mapcar
           #`(let ((,g!a #1=,(nth (car a1) places))
                   (,g!b #2=,(nth (cadr a1) places)))
               (if (,comparator ,g!b ,g!a)
                 (setf #1# ,g!b
                       #2# ,g!a)))
           (build-batcher-sn (length places))))))
```

small, fixed-size data-sets sorted, this macro is trivial to incorporate, sometimes even easier than the `sort` function. Instead of expanding into a lambda form, `sortf` expands into a tagbody form because `tagbody` is the canonical `progn` that returns `nil`. Here is an expansion of `sortf`:

```
* (macroexpand
    '(sortf < a b c))

(LET ()
  (TAGBODY
    (LET ((#:A1824 A) (#:B1823 C))
      (IF (< #:B1823 #:A1824)
        (SETF A #:B1823 C #:A1824)))
    (LET ((#:A1824 A) (#:B1823 B))
      (IF (< #:B1823 #:A1824)
        (SETF A #:B1823 B #:A1824)))
    (LET ((#:A1824 B) (#:B1823 C))
      (IF (< #:B1823 #:A1824)
        (SETF B #:B1823 C #:A1824)))))
T
```

The interface design of `sortf` is from *On Lisp*, but it is so natural that almost every lisp programmer would implement it this way. The first argument is usually a symbol representing a comparison operator—typically something like `<`. This usually

represents a function but, as Graham points out, could also represent a macro or special form since it is spliced directly into the function position of a list. We can even pass a lambda form since those are also allowed in the function position of a list[24]:

```
* (let ((a -3) (b 2))
    (sortf (lambda (a b) (< (abs a) (abs b)))
      a b)
    (list a b))

(2 -3)
```

Also like Graham's macro, the arguments to be sorted are generalised variables. That means that we can use `sortf` to sort any sort of variables, not only those represented by symbols, but anything that can be `setf`. Here is an example:

```
* (let ((a 2) (b '(4)) (c #(3 1)))
    (sortf < a (car b) (aref c 0) (aref c 1))
    (format t "a=~a b=~a c=~a~%" a b c))

a=1 b=(2) c=#(3 4)
NIL
```

While Graham's `sortf` and our `sortf` compile the same source language, their expansions couldn't be more different. Graham's macro is arguably more correct than ours because it will only execute the code to access the places once. With Graham's `sortf` we can pass in variables with side-effects and have them evaluated only once as might be expected. For example, Graham's `sortf` will only increment `i` once when given the place `(aref arr (incf i))`. Graham's `sortf` works by copying every variable to be sorted into a temporary binding, using a bubble sort to sort these temporary bindings, then using `setf` expansions[25] to write the temporary variables back to the

[24] Notice that we cannot pass a sharp-quoted lambda form though. Don't sharp quote your lambda forms.

[25] See On Lisp for an excellent description of the gory details on this.

original places, now in sorted order. Instead, our `sortf` will evaluate each place form many times throughout the sort, so you are advised to not use places that have side-effects. Another consequence of this design is that if you are after efficiency, make sure your accessors are efficient. In particular, do not use long list accessors like `caddr` because they will end up traversing the list many times. With our implementation, we sort the arguments *in-place*, that is without any temporary bindings. Instead of bubble sort, which has a *Big-O* complexity of `(O (expt N 2))`[26], we use Batcher's far better merge exchange sort with its `(O (* N (expt (log N) 2)))`. There are methods for constructing sorting networks of `(O (* N (log N)))`—the same as quick sort—but most use more operations for small network sizes than do Batcher's.

You will probably want to add a sharp-f fast declaration surrounding where you call `sortf` because it will not add this itself. If you want really fast sorts, ensure that the compiler knows the types of all your generalised variables being sorted. If you do specify types, always ensure that all the generalised variables are declared to be the same type. This is necessary because any element could potentially end up at any place.

But how do we know if this macro actually gives us any performance advantage over the `sort` function? We need to *benchmark* it. Benchmarking is discussed a lot because, especially to programmers, the timeless hobby of *bullshitting* is so enjoyable. Unfortunately, almost all benchmark results are useless. You are even advised to take the benchmark results from this book with a heavy grain of salt. That said, carefully crafted, controlled experiments that meter very slightly different versions of code running on the same lisp image on the same machine can sometimes be invaluable for understanding and fixing performance bottlenecks. Such metering is useful because not only can we tell which techniques are more efficient, but we can also tell just how much more efficient they are. Because they write code for us, macros are the best tool for setting up these experiments.

[26] We don't like infix.

Listing 7.45: SORT-BENCHMARK-TIME

```
(defmacro sort-benchmark-time ()
  `(progn
     (setq sorter (compile nil sorter))
     (let ((arr (make-array
                 n :element-type 'fixnum)))
       (time
         (loop for i from 1 to iters do
           (loop for j from 0 to (1- n) do
             (setf (aref arr j) (random n)))
           (funcall sorter arr))))))
```

The `sort-benchmark-time` macro is a component in our experiment. It expands into code that assumes either a lambda form or a function is bound to `sorter` and that this function will sort a fixnum array of size `n`. It then compiles this into a function and uses it to sort a randomly generated array `iters` iterations. The `time` macro is used to collect statistics on how long the sorting procedure takes.

`Do-sort-benchmark` is our actual interface for performing the benchmarking. Given a data-set size `n` and an iteration count `iters`, it will meter both the COMMON LISP `sort` function and our `sortf` macro. It preserves the state of the random number generator and resets it after performing the `sort` measurement but before running the `sortf` one so that the random arrays to be sorted are identical. It is important that `do-sort-benchmark` is compiled when run so that there is the least noise possible in our measurements.

When run, `do-sort-benchmark` tells us not only that `sortf` is efficient, but also that general purpose sorting algorithms are not even in the same ball-park as sorting networks with respect to performance for small, fixed-size data-sets. We also notice that `sortf` doesn't cons which, in turn, results in less garbage collection run-time, contributing even greater gains in performance. Here are the results for data-sets of sizes 2, 3, 6, 9, 25, 49:

```
* (do-sort-benchmark 2 1000000)
```

Listing 7.46: DO-SORT-BENCHMARK

```
(defun do-sort-benchmark (n iters)
  (let ((rs (make-random-state *random-state*)))
    (format t "CL sort:~%")
    (let ((sorter
           '(lambda (arr)
              #f
              (declare (type (simple-array fixnum)
                             arr))
              (sort arr #'<))))
      (sort-benchmark-time))

    (setf *random-state* rs)
    (format t "sortf:~%")
    (let ((sorter
           `(lambda (arr)
              #f
              (declare (type (simple-array fixnum)
                             arr))
              (sortf <
                ,@(loop for i from 0 to (1- n)
                        collect `(aref arr ,i)))
              arr)))
      (sort-benchmark-time))))

(compile 'do-sort-benchmark)
```

```
CL sort:
; Evaluation took:
;   1.65 seconds of real time
;   8,000,064 bytes consed.

sortf:
; Evaluation took:
;   0.36 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 3 1000000)

CL sort:
; Evaluation took:
;   3.65 seconds of real time
;   24,000,128 bytes consed.

sortf:
; Evaluation took:
;   0.46 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 6 1000000)

CL sort:
; Evaluation took:
;   10.37 seconds of real time
;   124,186,832 bytes consed.

sortf:
; Evaluation took:
;   0.8 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 9 1000000)

CL sort:
```

```
; Evaluation took:
;   19.45 seconds of real time
;   265,748,544 bytes consed.

sortf:
; Evaluation took:
;   1.17 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 25 1000000)

CL sort:
; Evaluation took:
;   79.53 seconds of real time
;   1,279,755,832 bytes consed.

sortf:
; Evaluation took:
;   3.41 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 49 1000000)

CL sort:
; Evaluation took:
;   183.16 seconds of real time
;   3,245,024,984 bytes consed.

sortf:
; Evaluation took:
;   8.11 seconds of real time
;   0 bytes consed.
```

So for certain tasks, order of magnitude or better improvements to our system sorting routines are possible with sorting networks. These measurements are not intended to make our sort implementation look bad (it is actually an excellent sorting

Listing 7.47: MEDIANF

```
(defun medianf-get-best-sn (n)
  (case n
    ((0)  (error "Need more places for medianf"))
    ((9)  paeth-9-median-sn)
    ((25) paeth-25-median-sn)
    (t    (prune-sn-for-median n
            (build-batcher-sn n)))))

(defmacro! medianf (&rest places)
  `(progn
     ,@(mapcar
         #`(let ((,g!a #1=,(nth (car a1) places))
                 (,g!b #2=,(nth (cadr a1) places)))
             (if (< ,g!b ,g!a)
               (setf #1# ,g!b
                     #2# ,g!a)))
         (medianf-get-best-sn (length places)))
     ,(nth (floor (1- (length places)) 2) ; lower
           places)))
```

routine), but rather to show a realistic example of how smart programming with macros can result in substantial efficiency gains. Lisp macros let us easily and portably program smart. Blub languages take so much effort to program smart that Blub programmers almost always settle for programming dumb. In lisp, everything compiles down to lisp so there are never any barriers to what you can optimise. If anything is ever unacceptably slow, change it and make it faster. We almost never need things to run fast, but when we do, lisp is the solution.

Another macro similar to `sortf` is `medianf`, which uses our pruned median selection networks or Paeth's hand-crafted median networks to sort the places just enough to ensure that the median element is in its correct position. In the case of even network sizes, both the lower and upper median will be in the correct place. Unlike `sortf`, which always returns `nil`, `medianf` will return the value of the lower median (which is the same as the upper median for odd network sizes).

As we said earlier, `sortf` and `medianf` sort any kind of places

you can `setf`. In the case of variables stored in registers, this gives lisp the opportunity to produce sorting code which doesn't even access memory. For example, here is the compiled expansion for `medianf` on three fixnum places:

```
* (dis ((fixnum a) (fixnum b) (fixnum c))
    #f
    (medianf a b c))
...
;;; (MEDIANF A B C)
      34:       MOV     EBX, EAX
      36:       CMP     EDX, EAX
      38:       JL      L4
      3A: L0:   MOV     EBX, EAX
      3C:       CMP     ECX, EAX
      3E:       JL      L3
      40: L1:   MOV     EAX, ECX
      42:       CMP     EDX, ECX
      44:       JNL     L2
      46:       MOV     ECX, EDX
      48:       MOV     EDX, EAX
      4A: L2:
...
      5B: L3:   MOV     EAX, ECX
      5D:       MOV     ECX, EBX
      5F:       JMP     L1
      61: L4:   MOV     EAX, EDX
      63:       MOV     EDX, EBX
      65:       JMP     L0
```

Lisp has more potential for efficient code than any other language and it's all thanks to macros. Because they are so good at creating controlled metering experiments, macros are also the solution to determining which techniques produce more efficient outcomes. Compilers are programs that write programs and macros are the best way to do that.

Chapter 8

Lisp Moving Forth Moving Lisp

8.1 Weird By Design

This chapter is a culmination of many macro techniques we have looked at so far in this book. Using macro abstractions we have developed, we create an implementation of one of my favourite programming languages: *forth.* Although this implementation embodies most of the important ideas of forth, it is very different and lispy. Though there are certain interesting uses for the code in this chapter, its primary purpose is to teach the concepts and fundamentals of forth meta-programming to a lisp audience and to be a platform for discussing the central theme of this book—creating and using duality of syntax with macros.

Forth, more so than every language except lisp, has a rich, fascinating history and I'm grateful for having discovered it. For that reason, and for everything else, this chapter is dedicated with love to my father Brian Hoyte who introduced me to forth and to computer programming. This chapter was partially inspired by [THREADING-LISP] and by the research of Henry Baker[LINEAR-LISP][LINEAR-LISP-AND-FORTH].

Forth was the first programming language that was created and developed without strong government, academic, or corpo-

rate sponsors—or at least the first such language to succeed. Instead of being motivated by the needs of a large organisation, forth was independently invented by Chuck Moore around 1968 to solve his own computing needs in astronomy, hardware design, and more. Since then, forth has been distributed, implemented, and improved upon by a passionate grass-roots user community[EVOLUTION-FORTH-HOPL2]. Contrast forth with the MIT (and later DARPA) patronage of early lisps and COMMON LISP, IBM's FORTRAN, and AT&T's unix language C.

Because of these roots, and because of a generally different philosophy of the role of computer software and hardware, forth is different. Even more so than lisp, forth looks *weird.* But like lisp, forth looks weird for a reason: it was designed with more in mind than style. Forth is weird by design, and this design relates to macros.

Today, forth is most commonly seen in so-called embedded platforms—computers that are severely resource constrained. It is a testament to the design of forth that the language can be entirely implemented on almost every programmable computer system ever created. Forth is designed to be as easy as possible to implement and experiment with. In fact, creating a forth clone is so profoundly trivial that inventing a forth-style stack based language or two is almost a rite-of-passage for programmers interested in the design of programming languages. Some stack based languages that can trace roots back to forth and have made interesting contributions are PostScript and Joy.

Often important forth implementation decisions are based on the exact resources of the computer that forth is being implemented on. Forth programmers have devised a set of *abstract registers* that need to be either mapped into real registers, mapped into memory locations, or possibly implemented in a different way altogether. But what do we do if we are implementing a forth on lisp, an environment with unlimited potential and few restrictions? Rather than simply impose an arbitrary mapping of forth abstract registers into lisp code, we try to take a step back. What would forth look like if Chuck had a lisp machine? Rather than fitting forth to the capabilities of an arbitrary machine, real or

Listing 8.1: FORTH-REGISTERS

```
(defvar forth-registers
        '(pstack rstack pc
          dict compiling dtable))
```

virtual, we explore a minimal set of forth abstract registers, optimised for simplicity and capability when implemented on lisp.

But really, searching for an optimal set of abstract concepts is what Chuck did while he created forth, using his experience of dozens of different forth implementations on as many architectures. This is why forth is so great. Like lisp, forth represents a high local maximum in the space of language design, and also like lisp, forth is not so much a programming language or a set of standards, but instead a building material and collection of wisdom regarding what works and what doesn't.

The `forth-registers` variable is a list of symbols that represent abstract registers for our forth machine. Of course lisp doesn't think in terms of registers and fixnums, but instead variables and symbols. It may seem strange to start our development of a forth environment here, with just a list of variable names, but this is in fact always the first step in implementing a forth system. Creating a forth is an ingenious process of bootstrapping exceeded in beauty and cleverness only by lisp. A modest description of this process follows.

One of the characteristic features of forth is its direct access to the stack data structures used by your program both to pass parameters to subroutines and to keep track of your execution path throughout these subroutines. Forth is especially interesting because—unlike most programming languages—it separates these two uses of the stack data structure into two stacks you can fool with[1]. In a typical C implementation, the parameters of a function call and its so-called *return address* are stored in a single, variable-sized *stack frame* for every function invocation. In forth, they are two different stacks called the parameter stack and the re-

[1]Most languages don't allow you to directly manipulate the stack at all.

Listing 8.2: FORTH-WORD

```
(defstruct forth-word
  name prev immediate thread)
```

turn stack, which are represented as our abstract registers `pstack` and `rstack`. We use the COMMON LISP `push` and `pop` macros, meaning these stacks are implemented with cons cell linked lists instead of the array data structures used in most forths.

The abstract register `pc` is an abbreviation for *program counter*, a pointer to the code we are currently executing. What forth code is and how we can point to it will be explained shortly, as will our abstract registers `compiling` and `dtable`.

Another building block of forth is its concept of a *dictionary*. The forth dictionary is a singly linked list of forth *words*, which are similar to lisp functions[2]. Words are represented with a lisp *structure*. Structures are efficient slot-based data structures usually implemented as vectors. The `name` slot is for a symbol used to lookup the word in the dictionary. Notice that the forth dictionary is not stored alphabetically, but instead chronologically. When we add new words we append them onto the end of the dictionary so that when we traverse the dictionary the latest defined words are examined first. The last element of our dictionary is always stored in the abstract register `dict`. To traverse the dictionary, we start with `dict` and follow the `prev` pointer of the word structures, which either point to the previously defined word or to `nil` if we are at the last word[3].

Given `w`, a word to lookup, and `last`, a dictionary to search, `forth-lookup` will return either a forth word structure or `nil` depending on whether the word `w` was found in the dictionary or not. The comparison function `eql` is used instead of `eq` because—unlike lisp—forth allows words to be named by numbers and other non-symbols.

[2]In other words, they aren't functions at all, but rather procedures.

[3]Yes, sometimes lisp programmers use alternatives to cons cells for linked lists.

Listing 8.3: FORTH-LOOKUP

```
(defun forth-lookup (w last)
  (if last
    (if (eql (forth-word-name last) w)
      last
      (forth-lookup
        w (forth-word-prev last)))))
```

The `immediate` slot of our forth word is a flag indicating whether the word is *immediate* or not. Immediacy is a forth meta-programming concept we will explore in depth shortly. For now here is a rough analogy to its lisp counterpart: immediate words are like lisp macros in that they are forth functions to be executed at compile time instead of run-time. What? Only lisp is supposed to have macros. While it is true that the COMMON LISP macro system is much more powerful than any other macro system—including the best forth implementations—forth has extension capabilities that surpass almost all other languages. Like lisp, this capability is the result of a design philosophy: if it's good enough for the language implementor, it's good enough for the application programmer. Like lisp, forth doesn't really recognise the notion of a primitive. Instead, it provides a set of *meta-primitives* that can be combined to build the language that you, the programmer, desire. Like lisp, and unlike most Blub languages, extending the language in novel ways through the use of macros is not only possible, but encouraged. Like lisp, forth is not about style, but instead, power.

8.2 Cons Threaded Code

In the previous section, we focused on abstract registers. These registers are an important focal point, and that is why forth philosophy considers them so fundamental, but these registers are actually just a component of a more general concept: *abstract machines*. Probably the most distinguishing property of different forth systems are their implementations of *threaded code*. What

forth means by threaded code is very different from the conventional meaning of preemptive scheduled, shared-memory processes[4]. Forth threads have nothing to do with concurrency. They are a framework for talking about code compilation and meta-programming.

While lisp gives access to the tree data structure[5] of symbols that your program is compiled from and to before being assembled into memory, forth provides no symbolic manipulation. Instead, forth gives access to this process of assembling—threading—the code into memory. Although to outsiders the most apparent features of forth are its stacks and postfix notation, it is actually threads that make forth what it is. Forth is about stacks in the same way that lisp is about lists. They just happen to be the most applicable data structures to use for solving meta-programming problems—what forth and lisp are both really about.

The classic style of threading is known as *indirect threaded* code but most modern forths are implemented with *direct threaded* code. The difference involves a level of indirection. The low-level efficiency implications of this indirection depend on the underlying processor and we will not get into details here. There are many good tutorials on forth threading[STARTING-FORTH][MOVING-FORTH]. In memory, these styles of threading both consist of adjacent *cells*, which are fixnum machine words representing pointers. A small piece of tight machine code called the *inner interpreter* is usually tailored for the processor being used because of its important job: to follow the pointers of these forth threads, interpreting their meanings as it goes along. The default behaviour when encountering a cell is to push the current program counter location onto the return stack and then point the program counter to whatever is contained in the cell. When the inner interpreter reaches the end of a thread, it pops the return stack and resumes execution at this location—where it left off[6].

As you can imagine, this type of program storage makes for

[4]This book does not recommend those types of threads for security and reliability reasons.

[5]Actually a directed acyclic graph.

[6]In most forths, the end of threads are indicated with the forth word `exit`.

extremely small programs. A compiled forth word is just a consecutive array of fixnums, most of which represent pointers to other words. This has always been one of the advantages of forth. Because of the transparency in the threading of the program into memory, forth allows fine control over many programming trade-offs, including one of the most important: Execution speed versus program size. Threaded code lets us optimise our abstractions as close to our problems as possible, resulting in extremely fast, small programs. But just as lisp macros are about much more than just efficiency, so are forth threads. As much as lisp programmers, forth programmers tend to think of themselves as implementors instead of mere users. Forth and lisp are both about control—making your own rules.

There are at least two other common types of forth threading techniques: *token threaded* code and *subroutine threaded* code. These represent opposite directions to take when considering the speed versus size trade-off. Sometimes these threading techniques coexist with indirect and direct threaded code in the same forth. Token threading involves adding another level of indirection by using fixnums even smaller than pointers to represent words in the thread. At the other end of the spectrum is subroutine threading. This type of threaded code is becoming popular and the best modern forth compilers partially use subroutine threading. Instead of consecutive pointers to words for the inner interpreter to follow, subroutine threaded code stores *inline* machine instructions to call these pointers. In subroutine threaded code, the inner interpreter disappears—it is actually implemented by the hardware (or virtual machine). Subroutine threaded code is usually considered an opaque block, one that only a special, non-programmable compiler can manipulate. Especially when various optimisations are made to the code, these opaque blocks start to look nothing like uniform, cell-based threads. Almost all non-forth compilers compile only to subroutine threaded code and don't imagine that you would ever want to do anything else, leading to this peculiar definition:

A *Flub* is a language that only considers subroutine

> threaded code or a language implementation that only provides subroutine threaded code.

For example, C is a Flub because it only provides programmers means to create functions—opaque blocks of subroutine threaded code. Certainly we can implement an inner interpreter in C to handle indirect threaded code[7] and bootstrap a stack-based language with this program, but then we are no longer programming in C. Almost all Blub languages are Flubs. Forth as an abstract machine is, as we've just described, not a Flub. As we will see, forth gives programmers/implementors lots of control over how their programs are compiled.

Is lisp a Flub? Interestingly, lisp was probably the first non-Flub programming language but has mostly turned into Flub. Although not strictly required to by the standard, most COMMON LISP compilers only compile functions to blocks of opaque machine code and, as such, are Flubs. But in very early versions of lisp, functions were stored as lists—a strange sort of code threading not entirely unlike forth threads. While this did allow some very clever run-time tricks, including giving meaning to cyclic code, it was hopelessly inefficient. Unlike forth's many types of threading—which have been efficiently implemented on almost all architectures—this internal representation for lisp functions was intolerable and lisp was changed to allow (extremely) efficient code. The consequence, for meta-programmers, is that most implementations of COMMON LISP are Flubs.

But there is a difference between features that are impossible to add to a language and features we can add with macros. With macros we can extend the language in any way we want and it remains lisp. COMMON LISP lacks threaded code in the same sense that it lacks continuations and first-class macros: they are omitted from the language deliberately and left for macro writers to implement as needed. One of the most important results of this chapter and its code is to show that even when they are Flubs, lisp languages can transform into non-Flub languages through macros. Non-Blub implies non-Flub, or, in other words, if you can't turn

[7] Direct threaded code is possible but more difficult.

Listing 8.4: FORTH-INNER-INTERPRETER

```
(defmacro forth-inner-interpreter ()
  '(loop
     do (cond
          ((functionp (car pc))
             (funcall (car pc)))
          ((consp (car pc))
             (push (cdr pc) rstack)
             (setf pc (car pc)))
          ((null pc)
             (setf pc (pop rstack)))
          (t
             (push (car pc) pstack)
             (setf pc (cdr pc))))
     until (and (null pc) (null rstack))))
```

a language into a non-Flub, it must be a Blub. However, the opposite is not true. Non-Flub languages like forth remain Blubs and the most straightforward way to turn them into non-Blubs currently known is to implement lisp environments with them—and then you're programming lisp.

Instead of using consecutive memory cells to represent threads as with indirect/direct threaded code, our forth takes advantage of lisp's dynamic typing and cons cell list structure. We call this *cons threaded* code. The macro `forth-inner-interpreter` expands into code that is capable of following these cons cell linked-list threads. It might seem strange to start programming the logic for our forth environment here—with a macro designed to be expanded into some as-of-yet unknown expression—but this is in fact a desirable lisp programming pattern. Because macros let us start programming anywhere we want, why not start with the really interesting, driving bits of a program? These are the bits that will have the most influence on the program's ultimate design.

The definition of `forth-inner-interpreter` is itself a concise definition of what we mean by cons threaded code. The car of every cons cell is points to either a function, another cons cell, or some other lisp atom. Functions are executed as they are en-

Listing 8.5: PRIM-FORMS

```
;; Prim-form: (name immediate . forms)
(defvar forth-prim-forms nil)

(defmacro def-forth-naked-prim (&rest code)
  `(push ',code forth-prim-forms))

(defmacro def-forth-prim (&rest code)
  `(def-forth-naked-prim
     ,@code
     (setf pc (cdr pc))))
```

countered. Notice that it is left for the function itself to update the pc register. If another cons cell is found in the thread it is assumed to indicate a subroutine call—a word invocation. Our inner interpreter will push the pc resume location onto the return stack and then jump to this new thread. If some other lisp atom is encountered, it is simply pushed onto the parameter stack and execution resumes at the next cell in our thread. The inner interpreter will return once it reaches the end of its thread and has no other threads to return to on its return stack.

But of course functions can't update the pc variable unless they are defined in its lexical scope[8] so we employ another macro technique: instead of using **defun**, we create a similar interface that does something completely different. **Def-forth-naked-prim** feels similar to creating **defun** defined functions except that the code it expands into pushes the user provided forms onto a list stored in **forth-prim-forms**. Our eventual macros will use these forms to define the forth primitives inside its lexical scope. Because these forms will always be expanded into this environment, we are free to write code that uses all of our forth abstract registers like pc, pstack, etc.

Primitives defined with **def-forth-naked-prim** will not update the pc variable to the next cons cell in the thread. For the majority of primitives we should use **def-forth-prim** so the usual

[8]Except see pandoric macros.

Listing 8.6: BASIC-PRIM-FORMS

```
(def-forth-prim nop nil)

(def-forth-prim * nil
  (push (* (pop pstack) (pop pstack))
        pstack))

(def-forth-prim drop nil
  (pop pstack))

(def-forth-prim dup nil
  (push (car pstack) pstack))

(def-forth-prim swap nil
  (rotatef (car pstack) (cadr pstack)))

(def-forth-prim print nil
  (print (pop pstack)))

(def-forth-prim >r nil
  (push (pop pstack) rstack))

(def-forth-prim r> nil
  (push (pop rstack) pstack))
```

update is performed. Both of these macros expect the first argument to be a symbol used to refer to the primitive and the second to be a boolean indicating whether the primitive is immediate or not. The remainder of the arguments are lisp forms to be evaluated when the primitive is execute.

Eight simple primitives—none of them naked or immediate—are presented now. **Nop** is a dummy instruction that does nothing ("no operation"). The * primitive is the multiplication operator: it pops the top two values from the parameter stack, multiplies them together, then pushes the result back. **Dup** is short for "duplicate" and pushes the top value on the parameter stack onto the parameter stack again leaving two duplicate values. **Swap** will exchange the top two parameter stack elements using a very useful COMMON LISP macro: **rotatef**. It is no coincidence that

forth also has (stack-based) rotation mechanisms. `Print` pops the parameter stack and prints it. `>r` transfers a value from the parameter stack to the return stack, `r>` does the opposite.

Does the name * violate our important variable capture rule from *section 3.5, Unwanted Capture* that forbids us from rebinding functions defined by COMMON LISP? No, because we haven't actually used this symbol to bind any functions—it's just the first element in one of the lists in `forth-prim-forms`. We have done nothing wrong. Symbols are independent from the functions or macros they are sometimes used to indicate. We can use any symbols anywhere, so long as we are careful to never violate our important variable capture rules. This only comes into play when writing lisp; we are writing forth.

8.3 Duality of Syntax, Defined

If you remember nothing else from this book, remember the message of this section. Here we finally define and explain a concept we have touched upon throughout: *duality of syntax*. This section assumes you have read at least the three introductory chapters, *chapter 6, Anaphoric Macros*, and the preceding forth sections.

To most lisp programmers the fact that programming in lisp is more productive and, eventually, more natural than programming in Blub is empirically obvious, but answering why this is the case is much more difficult. While it's true that lisp gets its amazing power of expression from macros—and we have seen many interesting ones in this book and elsewhere—all explanations so far feel unsatisfactory. What is the real advantage of macros? A partial explanation certainly includes *brevity*, making your programs short. Here is its definition:

> Let L be a programming language, F a feature in that programming language, and A an arbitrary program in L. F provides a brevity feature if A is shorter than it would be in a version of L without F.

Brevity features provide the basis and rational for the *theory of brevity*:

> The effort required to construct a program is inversely proportional to the amount of brevity features available in the programming language used.

The *theory of brevity* is based on the idea that if your programming abstractions make the expression of programs very short and concise, writing them becomes easier because less code needs to be written. Our CL-PPCRE read macros are examples of brevity features: they shorten the rather long CL-PPCRE function names into concise, Perl-style expressions that save us key-strokes every time we use them. The theory of brevity is very applicable to writing small programs for which we know where we want to go when we start[9]. Unfortunately, most programs aren't like this. Most programs—at least the interesting ones—are created *iteratively* through a series of interactive write-test cycles that take into account feedback at each step along the way. Your abstractions may be brief, but if you're always having to change them to different (perhaps equally brief) abstractions, you likely won't save much effort. Instead of considering the length of the final program, maybe we should consider the length of the process required to get there.

In every language, programs end up looking different from how they start. Most programs start with just a simple sketch that is filled out and detailed as the author learns more about the problem. Before we come back to brevity and duality, this chapter walks us through the development of a simple program that will motivate the discussion: our forth environment.

Hm, where were we? Ah yes, we had rambled on a lot about *abstract registers*, *abstract machines*, and *threaded code*, as well as defining a word lookup utility called `forth-lookup`, an inner interpreter for our cons threaded code, and a system for collecting lists representing *primitives* in our forth system. But what will forth on lisp be? Well, what is the most natural form for any abstraction that mixes behaviour and state? Closures, of course. Our old friends, let and lambda. Hacking up this idea might give the following macro:

[9] Often called Perl-one-liners, even when not written in Perl.

```
(defmacro new-forth ()
  `(let ,forth-registers
     (forth-install-prims)
     (lambda (v)
       (let ((word (forth-lookup v dict)))
         (if word
           (forth-handle-found)
           (forth-handle-not-found))))))
```

Our list of forth abstract registers, `forth-registers`, gets spliced directly into the expansion, initially binding all of the abstract registers to `nil`. Notice that we have left a lot of *holes* in the functionality of this macro. We have discovered that we are going to have to define a macro `forth-install-prims` which installs our primitive forms, as well as the macros `forth-handle-found` and `forth-handle-not-found`. But the most important thing we learn from this sketch is that, yes, this closure design looks like it could work. The idea, which came to us by just following the default lisp design, entails forth being a closure that is invoked once for every word we want to give it. Our sketch outlines an implementation for the following *use case*. Here we imagine creating a new forth environment:

```
(defvar my-forth (new-forth))
```

Here is some forth code for squaring the number 3 and printing the result:

```
3 dup * print
```

We could execute it on our forth environment like so:

```
(progn
  (funcall my-forth 3)
  (funcall my-forth 'dup)
  (funcall my-forth '*)
  (funcall my-forth 'print))
```

Listing 8.7: GO-FORTH

```
(defmacro! go-forth (o!forth &rest words)
  `(dolist (w ',words)
     (funcall ,g!forth w)))
```

Listing 8.8: FORTH-STDLIB

```
(defvar forth-stdlib nil)

(defmacro forth-stdlib-add (&rest all)
  `(setf forth-stdlib
         (nconc forth-stdlib
                ',all)))
```

Although this is a clumsy interface to use, we are programming lisp so we know we can always create a macro to hide these details, and that is exactly what is done with the **go-forth** macro. Notice that **go-forth** uses the automatic once-only functionality of **defmacro!** because the first argument provided to **go-forth** is inside a loop defined with **dolist** and will probably not be evaluated exactly once as might be intended by users of the macro. With **go-forth**, feeding forth code into our forth environment becomes much cleaner:

```
(go-forth my-forth
  3 dup * print)
```

At this point it might occur to us that we will eventually want to execute some forth bootstrapping code when creating new forth environments. So we need to be able to invoke the closure while creating it. This might require changing the program's let over lambda design or possibly creating some sort of wrapper function around our **new-forth** macro that uses the **new-forth** macro, loads in the standard library, and returns the resulting forth.

Because forth code is just a list of symbols and other atoms, our *standard library* that provides all of the bootstrapping we need

(except for a few more primitives) can be stored in a list. The variable `forth-stdlib` keeps this list of forth code to be executed when new forths are created and the `forth-stdlib-add` macro expands into lisp code that will concatenate new forth code onto the `forth-stdlib` list.

What is the easiest way to adapt `new-forth` to support loading this standard library? Do you remember the `alet` macro we wrote in *section 6.3, Alet and Finite State Machines*? The purpose of this macro was to create a duality of syntax with COMMON LISP's `let` while binding the anaphoric variable `this` around the provided code. `This` refers to the result that will be returned from `alet`—the forth closure.

So changing our sketch is even easier than expected. All we have to do is change the first `let` symbol in our sketch to an `alet` and then add some code to load the standard environment into `this`, the forth closure[10]. We didn't have to re-arrange anything because `alet`'s syntax was deliberately aligned with `let`'s. Here is what this next iteration looks like:

```
(defmacro new-forth ()
  `(alet ,forth-registers
     (forth-install-prims)
     (dolist (v forth-stdlib)
       (funcall this v))
     (lambda (v)
       (let ((word (forth-lookup v dict)))
         (if word
           (forth-handle-found)
           (forth-handle-not-found))))))
```

Remember that `alet` introduces a layer of indirection using a closure and therefore makes our forth environment slightly less efficient. However, just as we don't know if this efficiency burden will be too much, we also don't know we won't end up needing this indirection. To eliminate the indirection, use the `alet%` macro defined just prior to `alet`.

[10]Done after the primitives have been installed so that the standard library can make use of them.

Perhaps now, or perhaps later on when we're trying to build and debug our forth environment, it might occur to us that it would also be useful to be able to access the forth abstract registers from outside the forth environment. Unfortunately, these variables are closed over with a let over lambda. We will have to change our program again to make them accessible. There are, of course, many ways to do this. We could embed and return multiple closures in our forth environment, some of which could save and access the abstract registers, or we could re-consider our let over lambda strategy entirely. But before we do that, are there any dualities to help us? Remember `plambda` from *section 6.7, Pandoric Macros*? Its purpose was to create a duality of syntax with `lambda`, but one that creates closures that are actually open to the outside world. Changing our sketch to support this is a simple matter of prefixing a p character onto the `lambda` we return as our closure and adding the list of variables we want to export. Our list is conveniently available to us in `forth-registers`[11]. Our sketch becomes:

```
(defmacro new-forth ()
  `(alet ,forth-registers
     (forth-install-prims)
     (dolist (v forth-stdlib)
       (funcall this v))
     (plambda (v) ,forth-registers
        (let ((word (forth-lookup v dict)))
          (if word
            (forth-handle-found)
            (forth-handle-not-found))))))
```

With the forth closure opened up, we have available the following use case. This pushes five items onto a forth stack:

```
* (go-forth my-forth
    1 2.0 "three" 'four '(f i v e))
```

[11] Although splicing these variables in with unquote will only work when writing macros, read macros let us do similar things when writing functions.

Listing 8.9: NEW-FORTH

```
(defmacro new-forth ()
  `(alet ,forth-registers
     (setq dtable (make-hash-table))
     (forth-install-prims)
     (dolist (v forth-stdlib)
       (funcall this v))
     (plambda (v) ,forth-registers
       (let ((word (forth-lookup v dict)))
         (if word
           (forth-handle-found)
           (forth-handle-not-found))))))
```

```
NIL
```

And we can pandorically open **my-forth** to examine its parameter stack:

```
* (with-pandoric (pstack) my-forth
    pstack)

((F I V E) FOUR "three" 2.0 1)
```

This was the process performed to arrive at our final version of the macro **new-forth**. The final definition is identical to the last sketch except that it also sets the **dtable** abstract register to point to a hash-table (explained soon).

Programming, at least the interesting kind, is not about writing programs, but instead, changing them. In terms of productivity, brevity only takes us so far. We could rename **lambda** to, say, **fn**, but this brevity feature doesn't save much except for a few key-strokes here and there[12]. What does save us effort, however, is having lots of abstractions similar to lambda that we can use to change what code means without modifying the code itself much. Duality of syntax saves us effort.

[12]Many of us think **lambda** is just fine, thank you.

Just like putting *earmuffs* on your special variable names can bite you by forcing you to add or remove asterisks if you change your mind about whether a variable should be special or lexical[13], needlessly separating syntax and avoiding dualities can cause much pointless effort during programming. Another example: sharp quoting your lambda forms is a bad idea because it means you have just that much more to modify when you decide a function really needs to be an `alambda` or when you decide to use the lambda form in the function position of a list. Generalised variables also provide a very important duality: when writing macros, the same form can be spliced into expansions for both accessing and modifying the variable. COMMON LISP's dual meaning for the empty list and the false boolean value is yet another example—there is no real reason these two should be the same except for duality of syntax. Duality is also why this book has promoted closures instead of other CLOS features[14] such as `defclass` and `defmethod`. There is usually less *friction* when modifying programs that use closures than when modifying programs that use classes and objects because we have so many good dualities of syntax for closures and because programming macros that build closures is more uniform[15]. With these and other examples in mind, we can finally give a clear definition of what is meant by duality of syntax:

> Let L be a programming language, F a feature in that programming language, and A and B arbitrary programs in L. F provides a duality of syntax feature if the modifications required to change A into B become fewer than in a version of L without F.

Which leads to the *theory of duality*:

[13] Or possibly leaving incorrect documentation in code.

[14] Here the word "other" refers to the fact that even programming with closures is, in COMMON LISP, a feature of CLOS. CLOS is so fundamental that you can't escape it (nor would you want to).

[15] That said, generic functions are very important because they introduce dualities with multi-methods.

> The effort required to construct a program is inversely proportional to the amount of dual syntax available in the programming language used.

While the concept of duality of syntax and the impact of its benefits are both quite clear, how to actually design good dualities is much less so. What are the most useful dualities in a certain language? How can we tell which of two different languages will provide better dualities of syntax for some given problem?

Because with lisp we control the programming language completely, we can design our language with as much or little dual syntax as we please. Following this train of thought is, in my opinion, the most fruitful area of programming language research today. Using lisp macros, just how similar can we make all of our disparate programs to one another so that changing them into new programs becomes that much easier[16]?

In both the definitions of brevity and duality, whether the feature F is effective or not depends on the programs being written or changed. Sometimes features that provide brevities or dualities can, in certain situations, actually increase the effort required. The best approach may be to provide as many useful brevity and duality features as possible while removing the ones that end up being more trouble than they are worth.

8.4 Going Forth

In this section, we really get forth going by filling in the *holes* left in the **new-forth** macro from the previous section. After having verified that the forth threading mechanism works, we bootstrap a forth programming environment and, along the way, explain what forth *immediacy* is and how it relates to lisp macros.

In the definition of **new-forth**, we left a hole in the macro that is to be filled by **forth-install-prims**. We would like to use a named abstraction without throwing out our lexical environment, so it must be a macro. The point of this macro is to compile

[16]Sometimes changing programs into new programs is called development, especially when the resulting program is larger than the original.

Listing 8.10: FORTH-INSTALL-PRIMS

```
;; Prim-form: (name immediate . forms)
(defmacro forth-install-prims ()
  `(progn
     ,@(mapcar
         #`(let ((thread (lambda ()
                           ,@(cddr a1))))
             (setf dict
                   (make-forth-word
                      :name ',(car a1)
                      :prev dict
                      :immediate ,(cadr a1)
                      :thread thread))
             (setf (gethash thread dtable)
                   ',(cddr a1)))
         forth-prim-forms)))
```

and install the primitives into the forth dictionary when a new forth instance is created. `Forth-install-prims` expands into a `progn` form with each sub-form being an instruction to append a primitive word onto the `dict` linked list, wrap the provided code in a `lambda`, and set the word's `name` and `immediate` slots. In addition, the function created for each word by `lambda`, called `thread`, is added to our `dtable` hash-table (explained soon). Because all of these functions will be created in the scope of our original `new-forth` macro, they have full access to the forth environment specified by our abstract registers. Notice that the `thread` binding does not capture `thread` from any user provided code so we don't need to name it with a gensym.

We have said that forth provides a meta programming system not completely dissimilar from lisp's and that this system is based around a concept called *immediacy*. In traditional forths, there is a variable called `state` which is either zero or non-zero. If it is zero, the forth is considered to be in a regular interpreting (executing) state. If we give a word to forth in this state, that word will be looked up and executed. If, however, the `base` variable is non-zero, the forth is said to be in a compilation state. If we present a word to forth in this state, the address of the presented word

Listing 8.11: FORTH-PRIMS-COMPILATION-CONTROL

```
(def-forth-prim [ t ; <- t means immediate
  (setf compiling nil))

(def-forth-prim ] nil ; <- not immediate
  (setf compiling t))
```

is appended to the current thread being compiled—generally the most recently created word in the dictionary. There is one exception, however, and this is the important point about immediacy. If we are in compilation state and we are given an immediate word, that word will be executed instead of compiled. So, like lisp, forth allows us to execute arbitrary forth code at compile time.

Because we are building our forth as an abstract machine on lisp, we don't have to suffer arbitrary mappings of fixnum values to true and false. In lisp, we have a dynamic type system that lets us enjoy arbitrary mappings of all values to true and false. In place of the forth variable `state`, our forth system uses the `compiling` abstract register to store our compilation state as a lisp *generalised boolean*. The traditional forth words used to control the compilation state are [and], the open and close square brackets. [takes us out of compilation mode and thus necessarily must be an immediate word.] puts us back into compilation mode and so is only executed when we are interpret mode and doesn't have to be immediate. This choice of symbols may seem strange now but will become clearer in high level forth code. These square brackets allow us to designate a block of code to be executed in the middle of compiling a forth thread. In a certain sense, these brackets are like lisp's backquote and unquote operators. Here is how these words are usually used in forth code:

```
... compiled words ...
[ interpret these words ]
... more compiled words ...
```

Listing 8.12: FORTH-COMPILE-IN

```
(defmacro forth-compile-in (v)
  `(setf (forth-word-thread dict)
         (nconc (forth-word-thread dict)
                (list ,v))))
```

Like most of forth, these words are transparently specified which permits us use them in unusual ways. For example, these words aren't balanced in the same sense that lisp parenthesis are. If we choose, we can use them in the opposite direction:

```
... interpret these words ...
] compile these words [
... more interpreted words ...
```

We even have the appearance of nesting, but this is not really nesting because we only have a single boolean state: Compiling or not-compiling.

```
... compiled words ...
[ interpret these words
  ] compile these words [
  interpret these words
]
... more compiled words ...
```

Our forth uses the **forth-compile-in** macro as an abbreviation macro. This macro compiles a forth word into our current thread, the thread of the last word created. Because our threads are represented by cons cells, we can use the lisp function **nconc** to simply append a pointer to the destination word's thread onto our current thread.

Another hole we left in the **new-forth** macro was what forth should do if it was able to lookup a provided word in the dictionary. This hole is filled by the macro **forth-handle-found**. This macro implements forth immediacy as described above. If we're compiling and the word looked up is not immediate, we compile

Listing 8.13: FORTH-HANDLE-FOUND

```
(defmacro forth-handle-found ()
  `(if (and compiling
            (not (forth-word-immediate word)))
       (forth-compile-in (forth-word-thread word))
       (progn
         (setf pc (list (forth-word-thread word)))
         (forth-inner-interpreter))))
```

Listing 8.14: FORTH-HANDLE-NOT-FOUND

```
(defmacro forth-handle-not-found ()
  `(cond
     ((and (consp v) (eq (car v) 'quote))
        (if compiling
          (forth-compile-in (cadr v))
          (push (cadr v) pstack)))
     ((and (consp v) (eq (car v) 'postpone))
        (let ((word (forth-lookup (cadr v) dict)))
          (if (not word)
            (error "Postpone failed: ~a" (cadr v)))
          (forth-compile-in (forth-word-thread word))))
     ((symbolp v)
        (error "Word ~a not found" v))
     (t
        (if compiling
          (forth-compile-in v)
          (push v pstack)))))
```

it into our current thread. Otherwise we set our program counter pc to point to the thread of the looked up word and run the inner interpreter to execute the word. Recall that this macro will be expanded into a lexical environment in which **word** is bound to the looked-up forth word.

Our final hole in **new-forth** was what forth should do if it isn't able to find a word in its dictionary. **Forth-handle-not-found** fills this hole and implements some special cases. Recall that **forth-handle-not-found** will be expanded into a lexical environment that contains a binding **v** that refers to the value passed to forth.

We also know that if this code is invoked, v will not refer to any word in the dictionary. If v is a symbol, **forth-handle-not-found** will throw an error. If the value is not a symbol, the behaviour is to push v onto the parameter stack or, if we are compiling, to compile it into the current thread. Two special cases are checked for, however. If v is a list with the first element **quote**, we push the quoted value onto the parameter stack. This is so that we can push symbols onto the parameter stack without them being interpreted as words. The second special case is if v is a list with the first element **postpone**. **Postpone** is an ANSI Forth word that unites and clarifies a couple traditional forth words. **Postpone** is used to always compile a word, even if that word is immediate. So, if we are in compilation mode a postponed immediate word will be compiled into our current thread even though it is immediate. Here is an example of postponing the [word:

```
... compiling ...
(postpone [)
... still compiling ...
```

With all the holes filled in our **new-forth** macro we are now in a position to create new forth instances with the **new-forth** macro. Earlier we created a special variable called **my-forth** with **defvar**. Even if we hadn't, we could implicitly declaim it special along with assigning it a value with a top-level **setq**[17]:

```
* (setq my-forth (new-forth))

#<Interpreted Function>
```

We can now use forth with the **go-forth** macro:

```
* (go-forth my-forth
    2 3 * print)
```

[17]Some implementations prevent you from doing this. The solution is to upgrade to one that does (such as CMUCL) or to use **defparameter** instead of **setq** in such cases.

Listing 8.15: FORTH-PRIMS-DEFINING-WORDS

```
(def-forth-prim create nil
  (setf dict (make-forth-word :prev dict)))

(def-forth-prim name nil
  (setf (forth-word-name dict) (pop pstack)))

(def-forth-prim immediate nil
  (setf (forth-word-immediate dict) t))
```

```
6
NIL
```

But so far we have only defined the words **dup**, *, and **print**. To do anything useful we will need more primitives. Like lisp, production quality forth implementations have a large number of words defined for programmer convenience. Over decades of use, many common programming patterns have been identified, abstracted into words, and then introduced into common forth vernacular. Like lisp, having the ability to extend the language defined as part of the language has resulted in much valuable experimentation. Because it is this philosophy and process we are investigating, we will not define many forth words that experienced forth programmers rely on. Instead, we aim for the minimal set of primitives required to explain forth's meta-programming system so we can compare it to lisp macros.

Three more primitives are defined, none of which are immediate or naked: **create**, **name**, and **immediate**. The **create** word appends a nameless word to the dictionary. **Name** pops a value from the parameter stack and sets the name of the last word in the dictionary to this value. **Immediate** simply sets the last word defined to be an immediate word. By default, words are not immediate.

Recall that we can execute code with our **go-forth** macro on our **my-forth** forth environment. Here we square the number 3 and print the result:

```
* (go-forth my-forth
```

```
    3 dup * print)
```

```
9
```

Do we have enough of forth to start bootstrapping forth with forth words itself? Although we don't really have defining words yet, thanks to the transparent specification of threaded code, we can begin to write forth words using forth. For example, here we use **create** to append a new empty word to the dictionary:

```
* (go-forth my-forth
    create)
```

```
NIL
```

We now use] to begin compiling, add the words **dup** and * to the thread, then use [to take us out of compilation mode:

```
* (go-forth my-forth
    ] dup * [)
```

```
NIL
```

Now we have a new word in our dictionary—one with a complete forth thread that, when executed by our inner interpreter, will square the number on the top of the stack. But this word isn't very useful unless we have a way of accessing it. We can give it a name using the word **name**. The name we assign will be the key we use to access our new thread:

```
* (go-forth my-forth
    'square name)
```

```
NIL
```

Notice how the first value we passed to forth is quoted. Recall that we decided this behaviour should result in forth pushing the symbol **square** onto the parameter stack. This symbol is then consumed by the word **name**. Now that our word is named, we can evaluate it like any other word using the symbol **square**:

Listing 8.16: FORTH-START-DEFINING

```
(forth-stdlib-add
  create
    ] create ] [
  '{ name)
```

Listing 8.17: FORTH-STOP-DEFINING

```
(forth-stdlib-add
  { (postpone []) [
  '} name immediate)
```

```
* (go-forth my-forth
    3 square print)

9
NIL
```

So the general technique for creating new words is the following pattern:

```
create
] ... compiled words ... [
'whatever name
```

But we can use a bit of forth meta-programming to improve this interface. A definition of a new forth word, {, is added to the standard library. Its thread consists of two pointers, the first pointing to the word **create** and the second pointing to the word]. So when the thread for this word is executed, it will append a new word to the dictionary and flip us into compilation mode. Forth generally uses the : word for this purpose, but this conflicts with lisp's use of : so we have elected to use { to start word definitions.

Similarly, we add a complementary word } to the standard library (replacing the traditional forth ;). There is actually no reason to define this word—the only thing it does is take us out of compilation state. We already have the word [to do that for

us. Despite this, defining { is useful because it gives us *normal balanced brackets*[18] by creating a pair of words { and } that make defining new words intuitive.

We can now create a forth to take advantage of these new standard library features (throwing out our previous definition of the word square):

```
* (setq my-forth (new-forth))

#<Interpreted Function>
```

Here is what square looks like with the new word defining words { and }:

```
* (go-forth my-forth
    { dup * } 'square name)

NIL
* (go-forth my-forth
    5 square print)

25
```

And new threads can refer to our custom created words just as easily as primitives. Here is how we can define the word quartic as a thread with two pointers to our square word:

```
* (go-forth my-forth
    { square square } 'quartic name)

NIL
```

(Expt 1/2 4) is indeed 1/16:

```
* (go-forth my-forth
    1/2 quartic print)
```

[18] As opposed to the backwards balanced brackets forth's square brackets give for word defining.

```
1/16
NIL
```

Because non-symbols are directly compiled into the forth thread and our inner interpreter treats non-functions as data items to push onto the stack when encountered, we can include numbers into a word definition:

```
* (go-forth my-forth
    { 3 } 'three name
    three three * print)

9
NIL
```

Recall that we look up all elements passed to forth to see if they have been previously named in the dictionary using the **eql** function. The consequence of this is that we can use any lisp object to name a word. Here, we use a number[19]:

```
* (go-forth my-forth
    { 4.0 } '4 name
    4 4 * print)

16.0
NIL
```

Forth is an excellent language for learning how to use pointer scoping. Forth defines two simple operators that are used to read and write values from memory: @ (fetch) and ! (store). Because our forth words are stored in cons cells instead of memory words, dereferencing a pointer with fetch is implemented by taking the **car** of a pointer. Setting it with store is implemented by setting its **car** using **setf**. Fetch will pop a value from the parameter

[19]In many forths, common numbers like 0, 1, -1, etc are defined as words because a compiled reference to a word typically takes less memory than a compiled literal.

Listing 8.18: MEMORY-PRIMS

```
(def-forth-prim @ nil
  (push (car (pop pstack))
        pstack))

(def-forth-prim ! nil
  (let ((location (pop pstack)))
    (setf (car location) (pop pstack))))
```

stack, assume it is a cons cell, fetch its `car`, then push that on the stack. Store will pop a value from the parameter stack, assume it is a cons cell, pop another value from the stack, and store it into the first value's `car`. For example, here is how we can create and print a cyclic list:

```
* (let ((*print-circle* t))
    (go-forth my-forth
      '(nil) dup dup ! print))

#1=(#1#)
NIL
```

So now we're programming in forth using threaded code. Or are we? Did we ever leave lisp? The distinction between the two languages is so blurry that it can barely be discerned. The remainder of this chapter attempts to make this distinction even blurrier while explaining forth meta-programming further.

8.5 Going Forther

COMMON LISP has a lot of functions that we would like to be able to include in our forth threads. `Forth-unary-word-definer` expands into as many `def-forth-prim` forms as elements passed to its macro body. The elements are assumed to be symbols representing either functions or macros, but they could also be lambda forms. The only restriction with primitives named by lambda forms is that to invoke such primitives you will need to pass the

Listing 8.19: FORTH-UNARY-WORD-DEFINER

```
(defmacro forth-unary-word-definer (&rest words)
  `(progn
     ,@(mapcar
         #`(def-forth-prim ,a1 nil
             (push (,a1 (pop pstack))
                   pstack))
         words)))
```

Listing 8.20: FORTH-BINARY-WORD-DEFINER

```
(defmacro! forth-binary-word-definer (&rest words)
  `(progn
     ,@(mapcar
         #`(def-forth-prim ,a1 nil
             (let ((,g!top (pop pstack)))
               (push (,a1 (pop pstack)
                          ,g!top)
                     pstack)))
         words)))
```

same (eq) lambda form to the forth environment. Here is the expansion when passed one symbol, not:

```
* (macroexpand
    '(forth-unary-word-definer
      not))

(PROGN
  (DEF-FORTH-PRIM NOT NIL
    (PUSH (NOT (POP PSTACK))
          PSTACK)))
T
```

We can use any COMMON LISP function that accepts one argument and forth-unary-word-definer will define a forth primitive that applies this function to the top element of the forth parameter stack.

Listing 8.21: FORTH-AND-LISP-WORDS

```
(forth-unary-word-definer
  not car cdr cadr caddr cadddr
  oddp evenp)
(forth-binary-word-definer
  eq equal + - / = < > <= >=
  max min and or)
```

An extension on this idea is `forth-binary-word-definer` which does the same thing except for operators that accept two values. The forth convention of treating the second-to-top element as the first argument to binary functions like - and / is enabled by creating a temporary let binding to hold the top element of the parameter stack. Here is an expansion for the word -:

```
* (macroexpand
    '(forth-binary-word-definer
      -))

(LET ()
  (PROGN
    (DEF-FORTH-PRIM - NIL
      (LET ((#:TOP1767 (POP PSTACK)))
        (PUSH (- (POP PSTACK) #:TOP1767)
              PSTACK)))))
T
```

Exercise: When we use `forth-binary-word-definer`, how is it possible that we are able to treat macros like `and` and `or` as if they were first-class values?

Difficult exercise: Why was using a gensym (`g!top`) necessary for avoiding unwanted variable capture in `forth-binary-word-definer`? Hint: We've already discussed it in this section.

So these macros let us add various lisp functions to our forth primitive environment so we can use them from within forth. Here is an example use of a unary primitive, `cadr`:

```
* (go-forth my-forth
    '(a (b) c) cadr print)

(B)
NIL
```

And a binary one, <:

```
* (go-forth my-forth
    2 3 < print)

T
NIL
```

So far our forth threads have been *directed acyclic graphs*, that is they consist of cons cell structure that doesn't point to itself anywhere (isn't self-referential) and which eventually terminates at our primitives, the leaves of the tree. For example, we can use pandoric macros to get at the thread that we created in the previous section when we defined the `quartic` word:

```
* (with-pandoric (dict) my-forth
    (forth-word-thread
      (forth-lookup 'quartic dict)))

((#<Interpreted Function>   ;; square->|->dup
  #<Interpreted Function>)  ;;         |->*
 (#<Interpreted Function>   ;; square->|->dup
  #<Interpreted Function>)) ;;         |->*
```

The above comments only show it from the angle we printed the form in lisp. What we can't see from the code or the comments is that this thread structure is actually shared. To see that, use `eq`:

```
* (eq (car *) (cadr *))

T
```

Or look at it with ***print-circle*** set:

```
* (let ((*print-circle* t))
    (print **)
    t)

(#1=(#<Interpreted Function>  ;; square->|->dup
     #<Interpreted Function>) ;;         |->*
 #1#)                         ;; --------|
T
```

Threaded code can allow forth amazing memory and size advantages. Entire forth systems are compiled code that is threaded together like this—from the network drivers to the highest level user programs. What's more, notice that we can cleanly extract the thread from **quartic** without taking a lot of extraneous other threads. For instance, we have many more primitives in our language, such as **+** and **cadddr** but they don't appear at all in the thread above. It's almost like we have a mark-sweep garbage collection algorithm that extracts only the threads that will be needed to execute a given forth word. In lisp this process is called *tree shaking* and is generally not very effective. In forth, however, it is extremely effective.

Unfortunately, the **quartic** thread we pandorically extracted from **my-forth** is not really that useful to us. It still permanently resides in the **my-forth** closure. That is, the lambda expressions representing the **dup** and ***** primitives have had their references to the forth abstract registers captured by an expansion of our macro, **new-forth**. Can we pull this code back up to the lisp macro surface so as to embed it in new programs? We will return to this soon but first we go a bit further into forth meta-programming.

At some level in any language—usually a level hidden from the programmer—it is necessary for code to be able to refer to itself. The most convincing example of this necessity is the observation that code needs to be able to refer to itself somehow in order to implement looping, recursion, and conditional expressions like **if** statements. The difference between *Flub* languages and non-Flub languages is that Flub prevents you from directly customising how

Listing 8.22: BRANCH-IF

```
(def-forth-naked-prim branch-if nil
  (setf pc (if (pop pstack)
               (cadr pc)
               (cddr pc))))
```

and where self-references are inserted. But, as we are doing now, lisp's non-Blub status means we can make it a non-Flub.

Our forth system in its current state (without it being able to insert self-references) is almost a *pure Flub*. In a similar sense as to how pure functional languages deliberately define a language to be devoid of side-effects and non-static mappings, pure Flub languages are defined to be devoid of self-referential code constructs like loops and recursion. A consequence of this is that interpreting pure Flub threads will always terminate. Our forth environment is not completely pure because we can—and will—violate this, but is pure in the sense that if only used as so-far described will result in pure Flub threads. Pure Flub is not very useful so let's ruin the Flub purity of our forth environment. Instead of going in the Flub direction—towards Flub languages like COMMON LISP where code threading is opaque and inaccessible—let's go in the direction of forth and make this attribute of code macro customisable.

The `branch-if` primitive is the first *naked primitive* presented so far. Recall that naked primitives are primitives that don't update the program counter abstract register (`pc`) automatically. Instead, they must update it themselves. `Branch-if` will pop a value off the parameter stack. If the value is non-null, `pc` is set to the contents of the next cell in the thread being interpreted. If the value is `nil`, `pc` is resumed as usual except that it skips over the next cell in the thread being interpreted.

For example, the following creates a forth environment so we can take advantage of our new `branch-if` primitive, and defines two words: `double` and `if-then-double`.

```
* (go-forth (setq my-forth (new-forth))
```

```
    { 2 * } 'double name
    { branch-if double "Not doubling" print }
        'if-then-double name)

NIL
```

Double simply multiplies the top element of the parameter stack by two, doubling it. **If-then-double** requires two items on the parameter stack. The top element is consumed and the second from top element being doubled only if the top element was non-null. Notice that because the next value in the thread after **branch-if** is a pointer to another thread (**double**), control of execution is transfered to this other thread without pushing a resume location onto the return stack. In lisp this is called a *tail-call.* So if we pass **nil**[20] to **if-then-double** then the branch is taken, no doubling happens, and the string is printed:

```
* (go-forth my-forth
    4 'nil if-then-double print)

"Not doubling"
4
NIL
```

But if the value is non-null, the branch is not taken, the doubling does happen, and the string is not printed:

```
* (go-forth my-forth
    4 't if-then-double print)

8
NIL
```

There is an easier way to exit from a word though, and it is implemented with a new word called **exit**. An interesting property of forth is that the word being called can decide whether or not

[20]We need to quote **nil** because we don't want it to be looked up in the forth dictionary.

Listing 8.23: EXIT

```
(forth-stdlib-add
  { r> drop } 'exit name)
```

Listing 8.24: COMPILER-PRIMS

```
(def-forth-naked-prim compile nil
  (setf (forth-word-thread dict)
        (nconc (forth-word-thread dict)
               (list (cadr pc))))
  (setf pc (cddr pc)))

(def-forth-prim here nil
  (push (last (forth-word-thread dict))
        pstack))
```

it was a tail-call. **Exit** is a normal forth word and so is invoked as usual: forth pushes the current thread location onto the return stack and then sets the program counter to point to the start of the **exit** word. When **exit** is invoked, because it has direct access to the return stack with the primitives **r>** and **>r**, we can cause the calling word to never get back control of execution by simply removing the resume location from the return stack and dropping it out of existence. Here is an example use of **exit**:

```
* (go-forth my-forth
    { "hello" print
      exit
      ;; Never gets here
      "world" print } 'exit-test name

    exit-test)

"hello"
NIL
```

So **branch-if** implements a jump, or a *goto* instruction, possibly jumping to the value stored in the subsequent cell of the

thread we are currently executing. Taking values from the thread you are currently executing is a common pattern in forth and requires naked primitives. Another primitive, `compile`, also uses this pattern. `Compile` is a naked primitive that will take the value of the next cell in the thread currently executing and then compile this value into the thread of the last word added to the dictionary—usually the word we are currently compiling. `Here` is a simple primitive that pushes the last cons cell of the thread being compiled onto the parameter stack. Our `here` is slightly different from the regular forth word `here`. In forth, `here` normally pushes the next location that will be compiled to instead of the last location compiled. This is because, in traditional forth, the memory location to be compiled next is known at this time—it will be the next adjacent memory cell. With cons threaded code we can't know this because we have not yet consed that memory.

With `compile` and `here` available we can now begin to write forth macros. Remember that when a forth word is *immediate*, at compile-time it will be executed instead of being compiled into the thread of the last word defined. In similar ways to how we can write macros to adapt and extend lisp, we can use immediate words to adapt and extend forth. In lisp, the basic data structures used for meta-programming are lists. In forth, the basic data structures are stacks.

You might have noticed that our forth environment doesn't even provide an if statement. We have a conditional branching primitive called `branch-if`, but so far this has only been useful for making tail-calls to other forth words. Recall that forth words are represented by threads and we can put a value for any thread into the cell jumped to by the `branch-if`. What if we put a value that leads to a part of the thread currently being compiled? We would have, in a sense, a tail-call to another part of our current forth word. Well, if statement is exactly that—a conditional tail call to the end of the if statement that is taken only when the condition is null.

Because we are programming completely in forth now, there is no need to add new primitives. To add an if statement to forth, we merely append some forth code to our standard library

Listing 8.25: IF

```
(forth-stdlib-add
  { compile not
    compile branch-if
    compile nop
    here } 'if name immediate)
```

with the **forth-stdlib-add** macro. Notice that **if** is defined as an immediate word, meaning that it should only be used when compiling. But since it is immediate, it will be executed, not compiled. When immediate words are encountered, nothing is automatically compiled into the target thread. So **if** itself compiles in three words to the target thread: **not**, **branch-if**, and **nop**. It then executes the word **here** which leaves the address of the last compiled word (the **nop**) on the stack. It leaves it on the stack? That is a strange thing for a word to do at compile time. What stack does it leave it on? Technically, the stack in use at compile time is called the *control stack*. In most forths the control stack is one and the same as the parameter stack. The distinction is necessary because of the variety of ways that forth can be implemented. Sometimes, especially in cross-compilation environments, the control stack is completely separate from what will eventually be the parameter stack. But here—as with most interactive forth environments—we use the parameter stack for the control stack.

So, **if** pushes a value corresponding to a location where a **nop** was compiled. What use is this? The **nop** itself is not very important, but instead what precedes it. In the cell immediately before the **nop** there is a **branch-if** instruction compiled in. Whatever we change the value of the **nop** to be will be the location our inner interpreter branches to if the **if** condition turns out to be null.

But why did we put a **nop** there instead of the memory address? It's because we don't yet know the memory address. We need to wait for the programmer to execute another immediate word—**then**—which will consume the value left on the control stack by **if**. **Then** will compile in a **nop** itself and write the loca-

Listing 8.26: THEN

```
(forth-stdlib-add
  { compile nop
    here swap ! } 'then name immediate)
```

Listing 8.27: ABS

```
(forth-stdlib-add
  { 0 swap - } 'negate name
  { dup 0 < if negate then } 'abs name)
```

tion of this **nop** over the **nop** compiled in by **if**. So all the words between the **if** and the **then** will be skipped over if the condition is null.

Abs is a word that makes use of **if** and **then** to calculate the absolute value of the top item on a stack. It simply checks if the value is below 0 and, if so, it calls another word, **negate**, to convert the negative value into its absolute value.

The most important reason why the control stack is used in this compilation process is because by using a stack it is possible to have control structures like if statements *nest*. That is, we can include if statements inside other if statements as long as we make sure to balance all **if** words with corresponding **then**s.

Because forth is a non-Flub language, how such threads are created and threaded together with control structures like if statements is *transparently specified* and open for us to adapt and

Listing 8.28: ELSE

```
(forth-stdlib-add
  { compile 't
    compile branch-if
    compile nop
    here swap
    compile nop
    here swap ! } 'else name immediate)
```

Listing 8.29: MOD2

```
(forth-stdlib-add
  { evenp if 0 else 1 then } 'mod2 name)
```

Listing 8.30: BEGIN-AGAIN

```
(forth-stdlib-add
  { compile nop
    here } 'begin name immediate
  { compile 't
    compile branch-if
    compile nop
    here ! } 'again name immediate)
```

extend. Most languages have an else clause associated with if statements; maybe we should add one too. Another immediate word **else** is added to the standard library. **Else** compiles into an unconditional branch to the terminating **then** so that if we took the true (secondary or consequent) branch, we will skip the false (tertiary or alternant) branch. **Else** then makes use of the value left on the stack by **if** to replace this **nop** with the location of the start of the else clause. It then leaves the location of its own **nop** on the stack for **then** to make use of. Because the behaviour we want **then** to perform is the same whether the location on the control stack was left by **if** or by **else**, **then** still works even if there is no **else** clause.

The word **mod2** uses **if**, **else**, and **then** to reduce an integer to its natural residue under a modulus of 2. It pushes a 0 if the top of stack is even or a 1 if it is odd.

Because our conditionals perform tail-calls to other parts of the thread being compiled, there is no reason why we can't use the exact same techniques to create iteration constructs like loops. The most basic forth loop is defined by the **begin** and **again** immediate words. These two words provide a simple infinite loop and are implemented very similarly to **if** and **then**, except that the address saved on the control stack between seeing these two

words corresponds to an address that should be compiled into a branch statement—not a location to compile an address. Here is a simple loop that counts down to 1 from a number provided on the stack and then exits from the word:

```
* (go-forth my-forth
    { begin
        dup 1 < if drop exit then
        dup print
        1 -
      again } 'countdown name

    5 countdown)

5
4
3
2
1
NIL
```

Notice in the above example that an **if** and **then** construct is nested inside the **begin-again** loop. Thanks to forth's control stack it is perfectly acceptable to nest any control structures that respect the stack. To respect the stack, a control structure should avoid popping values it didn't push and should avoid leaving any extra values when finished. But just like we often choose to violate referential transparency when constructing lisp macros, in forth we often choose to not respect the stack when compiling. The following example is identical to the previous, except we don't use the word **exit** to escape from the loop. Instead, we flip into compile mode with the [and] words and swap the pointers placed there by **if** and **begin** so that we can use their matching **then** and **again** words out of order:

```
* (go-forth my-forth
    { begin
      dup 1 >= if
```

```
                dup print
                1 -
                [ swap ] again
              then
    drop
} 'countdown-for-teh-hax0rz name

5 countdown-for-teh-hax0rz)
```

```
5
4
3
2
1
NIL
```

In the above, the code compiled by **again**, which is the code to bring us back to **begin**, is only executed inside the if statement. Very few other languages allow you access to the compiler in this way—only non-Flub languages to be precise. Because of this freedom, forth programmers are sometimes even more accustomed to macro *combinations* than are lisp programmers. Although the lisp code in this book uses macro combination techniques regularly, these techniques, and the leverage they can enable, aren't exploited to nearly their full possible extent by most existing lisp code. However, as this book has tried to illustrate, lisp is exceptionally well-suited to macro combinations. Such combination techniques are where I think some of the biggest wins in programmer productivity will be found in the next decade or so of language research.

8.6 Going Lisp

So far this chapter has defined a minimalist forth environment and demonstrated some of the most important forth metaprogramming concepts from a lispy perspective. Hopefully it has shown just how little effort it can take to design and implement

Listing 8.31: PRINT-FORTH-THREAD

```
(defun get-forth-thread (forth word)
  (with-pandoric (dict) forth
    (forth-word-thread
      (forth-lookup word dict))))

(defun print-forth-thread (forth word)
  (let ((*print-circle* t))
    (print (get-forth-thread forth word))
    t))
```

a forth when you have the right tool (COMMON LISP). We can write forth programs to write forth programs—but we already knew that. That's what forth is all about. Further, because of lisp's macro system we can write lisp programs to write forth programs. But can we write forth programs to write lisp programs?

Recall that our forth threads are cons cells linked together and that the leaves of these trees are either functions (representing primitives) or atoms (representing data to push onto the parameter stack). Because we decided to make the forth abstract registers accessible through pandoric macros, writing utilities to acquire and print forth threads is easy. `Get-forth-thread` pandorically opens the forth closure `forth` passed to it then retrieves and returns the thread of the word given in `word`. `Print-forth-thread` prints this resulting thread with `*print-circle*` bound to `t` in case it contains cycles.

To demonstrate, suppose we define two forth words `square` and `square3`:

```
* (go-forth my-forth
    { dup * } 'square name
    { 3 square print } 'square3 name)

NIL
```

In the compiled forth thread, all the symbols and other word information has been removed. All we have is a piece of list struc-

ture extracted from the dictionary of the forth `my-forth`:

```
* (print-forth-thread my-forth 'square3)

(3
 (#<Interpreted Function>
  #<Interpreted Function>)
 #<Interpreted Function>)
T
```

The above code has no cycles and is thus a pure-Flub program. As said earlier, almost all interesting programs contain cycles. To create conditionals and loops we can use the naked forth primitive `branch-if` which allows us to change the `pc` abstract register to point somewhere indicated by the value in the subsequent cell in the forth thread being executed. We also were able to implement tail calls by directly accessing the return stack with `>r` and `r>`. Unlike most other languages, we can directly customise which calls are tail calls—even from inside the word being called.

But it seems we're missing one construct of central importance to lisp: recursion. Can a word call itself? We saw how we can use `branch-if` to jump back to the start of a word—tail recursion. What we would really like to do, however, is have a word call itself through the usual thread mechanism. To do this, it must have the start of its thread location stored as a cell in our thread so the current location is stored on the return stack and then it must set `pc` to the start of the word. So far none of our words has been able to make use of *full recursion*, however, because we don't name the word until we are done compiling it—it is inaccessible to us when we search the dictionary trying to compile a recursive call. Luckily, there is a simple trick we can use to get around this. We can simply flip out of compile mode and name the word we are compiling before we compile a recursive call. Here is an example definition of a fully recursive factorial calculator:

```
* (go-forth (setq my-forth (new-forth))
    { [ 'fact name ]
      dup 1 -
```

```
      dup 1 > if fact then
      * })

NIL
```

Sure enough, (fact 5) is 120:

```
* (go-forth my-forth
    5 fact print)

120
NIL
```

Exercise: Some forths use a word recurse which simply looks up the thread of the word currently being compiled and inserts it into the thread being compiled. This is called *anonymous recursion.* Write an immediate word that does this as an alternative to the above trick for implementing *named recursion.*

Fact's thread is more complicated than square3 above. It contains self-referential code:

```
* (print-forth-thread my-forth 'fact)

#1=(#2=#<Interpreted Function>
    1 #<Interpreted Function> #2# 1
    #<Interpreted Function>
    #<Interpreted Function>
    #<Interpreted Function>
    #4=(#<Interpreted Function>
        #<Interpreted Function>)
    #1# . #4#)
T
```

In the above, #2# refers to the dup primitive and was compiled twice. #1# refers to the fact thread itself, implementing the recursion.

These structures look a lot like the lisp list structure that we use to write lisp programs, don't they? Because we understand

the abstract machine that will execute these threads, we can, with a few restrictions explained shortly, compile these forth threads back into lisp list structure that can be inserted into an expression by a macro and compiled with our lisp compiler. This process is known as *flubifying* the code because we convert the compiled program from a uniform, programmable data structure (a thread) into an opaque, inaccessible block of code (a compiled COMMON LISP function).

There are, of course, major differences between forth threads and lisp list structure we can evaluate or insert into macros. First, the forth primitives are simple pointers to functions (displayed here as `#<Interpreted Function>`), but we need the lisp list structure that these functions were created with. Now is finally the time to explain the `dtable` abstract register we created. `Dtable` is a hash-table that gives a mapping from these functions to the list structure that created them, populated when the forth is created.

A large difference between forth threads and our lisp programs is that forth threads assume they can make use of a return stack—a concept that doesn't really exist in Flubs like COMMON LISP. We want to remove the need for our `inner-interpreter` code and, instead, let the lisp compiler handle this with regular lisp control structures like function calls and `tagbody/go` forms.

The remaining code in this chapter is presented differently than most of the rest of the code in this book in that its implementation is not described in detail, but rather from a high-level perspective. This is because the mechanics of the implementation are complicated and messy and, honestly, not that interesting. Suffice it to say that I suspect most lisp programmers would implement it similarly.

`Flubify-aux` is macro that expands into a function that takes a forth thread and converts it to a flubbed piece of lisp code, taking advantage of the fact that every non-primitive word is surrounded by a `tagbody` and so gensyms can be used as tags for gotos.

`Assemble-flub` is used heavily through `flubify-aux` as an abbreviation that checks a hash-table `go-ht` to see if there were any `go`s found on a previous pass that reference the location we are

Listing 8.32: FLUBIFY-AUX

```
(defmacro flubify-aux ()
  `(alambda (c)
     (if c
       (cond
         ((gethash (car c) prim-ht)
           (assemble-flub
             `(funcall
                ,(gethash (car c) prim-ht))
             (self (cdr c))))
         ((gethash (car c) thread-ht)
           (assemble-flub
             `(funcall #',(car (gethash (car c)
                                        thread-ht)))
             (self (cdr c))))
         ((eq (car c) branch-if)
           (assemble-flub
             `(if (pop pstack)
                (go ,(gethash (cadr c) go-ht)))
             (self (cddr c))))
         ((consp (car c))
           (flubify forth (car c) prim-ht
                    thread-ht branch-if)
           (self c))
         (t
           (assemble-flub
             `(push ',(car c) pstack)
             (self (cdr c)))))))))
```

Listing 8.33: ASSEMBLE-FLUB

```
(defmacro assemble-flub (form rest)
  `(if (gethash c go-ht)
     (list* (gethash c go-ht)
            ,form
            ,rest)
     (list* ,form
            ,rest)))
```

Listing 8.34: FLUBIFY

```
(defun flubify (forth thread prim-ht
                thread-ht branch-if)
  (unless #1=(gethash thread thread-ht)
    (setf #1# (list (gensym)))
    (let ((go-ht (make-hash-table)))
      (funcall
        (alambda (c)
          (when c
            (cond
              ((eq (car c) branch-if)
                (setf (gethash (cadr c) go-ht)
                  (gensym))
                (self (cddr c)))
              ((consp (car c))
                (flubify forth thread prim-ht
                         thread-ht branch-if)))
              (self (cdr c))))
        thread)
      (setf #1# (nconc #1# (funcall
                            (flubify-aux)
                            thread))))))
```

currently compiling. If they are, it adds the gensym tag chosen earlier for it to the tagbody form.

`Flubify` is the function that uses `flubify-aux`. It does the first pass, checking for `branch-if` instructions and building up the `go-ht` hash-table. It also recursively flubs all threads that are connected to our current thread. In fact, `flubify` can actually be *doubly recursive*—you just can't see it until you expand the use of `flubify-aux`. You can't see it, but lisp can. If referential transparency is a transparent pane of glass, here we are looking into a house of mirrors.

`Compile-flubified` takes a hash-table built by `flubify` and converts it into a form that uses `labels` to bind each of these flubbed threads into a function named in the function namespace by a gensym. Inside this scope, its expansion then `funcalls` the original thread (which is also flubbed).

Listing 8.35: COMPILE-FLUBIFIED

```
(defun compile-flubified (thread thread-ht)
  `(labels (,@(let (collect)
                (maphash
                  (lambda (k v)
                    (declare (ignore k))
                    (push
                      `(,(car v) ()
                         (tagbody ,@(cdr v)))
                      collect))
                  thread-ht)
                (nreverse collect)))
     (funcall #',(car (gethash thread thread-ht)))))
```

Listing 8.36: FLUBIFY-THREAD-SHAKER

```
(defun flubify-thread-shaker
       (forth thread ht tmp-ht branch-if compile)
  (if (gethash thread tmp-ht)
    (return-from flubify-thread-shaker)
    (setf (gethash thread tmp-ht) t))
  (cond
    ((and (consp thread) (eq (car thread) branch-if))
       (if (cddr thread)
         (flubify-thread-shaker
           forth (cddr thread) ht
           tmp-ht branch-if compile)))
    ((and (consp thread) (eq (car thread) compile))
       (error "Can't convert compiling word to lisp"))
    ((consp thread)
       (flubify-thread-shaker
         forth (car thread) ht
         tmp-ht branch-if compile)
       (if (cdr thread)
         (flubify-thread-shaker
           forth (cdr thread) ht
           tmp-ht branch-if compile)))
    ((not (gethash thread ht))
       (if (functionp thread)
         (setf (gethash thread ht)
           (with-pandoric (dtable) forth
             (gethash thread dtable)))))))
```

Listing 8.37: FORTH-TO-LISP

```
(defun forth-to-lisp (forth word)
  (let ((thread (get-forth-thread forth word))
        (shaker-ht (make-hash-table))
        (prim-ht (make-hash-table))
        (thread-ht (make-hash-table))
        (branch-if (get-forth-thread forth 'branch-if))
        (compile (get-forth-thread forth 'compile)))
    (flubify-thread-shaker
      forth thread shaker-ht
      (make-hash-table) branch-if compile)
    (maphash (lambda (k v)
               (declare (ignore v))
               (setf (gethash k prim-ht) (gensym)))
             shaker-ht)
    (flubify forth thread prim-ht thread-ht branch-if)
    `(let (pstack)
       (let (,@(let (collect)
                 (maphash
                   (lambda (k v)
                     (push `(,(gethash k prim-ht)
                              (lambda () ,@(butlast v)))
                           collect))
                   shaker-ht)
                 (nreverse collect)))
         ,(compile-flubified
             thread thread-ht)))))
```

Flubify-thread-shaker is the function that actually traverses our forth thread. It recursively shakes all connected forth threads. This means that it isolates just the relevant forth thread structure necessary to execute the given thread with the **get-forth-thread** utility, then translates each function into corresponding lisp code, skipping over **if-branches** and erroring on seeing a **compile**.

Forth-to-lisp is the ultimate function that the earlier macros and functions of this chapter facilitate. It takes a forth environment created by **new-forth**, looks up the thread indicated by the symbol passed as **word**, then returns corresponding lisp code to execute this thread. It first shakes the thread (recursively shaking all its connected threads), then applies the flubification procedure.

Finally, it wraps a small amount of lisp code which implements the inner interpreter with regular lisp control structures.

To demonstrate, recall the forth words **square** and **square3** we defined earlier. Again, here is how they were defined in the **my-forth** forth environment:

```
* (go-forth my-forth
    { dup * } 'square name
    { 3 square print } 'square3 name)

NIL
```

Here we convert the **square3** thread to lisp:

```
* (forth-to-lisp my-forth 'square3)

(LET (PSTACK)
  (LET ((#:G1814 (LAMBDA () ; dup
                   (PUSH (CAR PSTACK) PSTACK)))
        (#:G1815 (LAMBDA () ; *
                   (PUSH (* (POP PSTACK)
                            (POP PSTACK))
                         PSTACK)))
        (#:G1816 (LAMBDA () ; print
                   (PRINT (POP PSTACK)))))
    (LABELS ((#:G1817 () ; square3
               (TAGBODY
                 (PUSH '3 PSTACK)
                 (FUNCALL #'#:G1818)
                 (FUNCALL #:G1816)))
             (#:G1818 () ; square
               (TAGBODY
                 (FUNCALL #:G1814)
                 (FUNCALL #:G1815))))
      (FUNCALL #'#:G1817))))
```

Sure enough, the above is executable lisp code. If we wanted, we could compile it in somewhere using a macro. Or we could just **eval** it:

```
* (eval *)

9
NIL
```

To show how a forth thread with both branching and recursion is flubbed, here is a portion of the compiled lisp code from the forth word **fact**:

```
* (forth-to-lisp my-forth 'fact)
    ...
    (LABELS ((#:G1803 ()                    ; fact
               (TAGBODY
                 (FUNCALL #:G1797)          ; dup
                 (PUSH '1 PSTACK)
                 (FUNCALL #:G1798)          ; -
                 (FUNCALL #:G1797)          ; dup
                 (PUSH '1 PSTACK)
                 (FUNCALL #:G1799)          ; >
                 (FUNCALL #:G1800)          ; not
                 (IF (POP PSTACK) (GO #:G1804))
                 (FUNCALL #'#:G1803)        ; fact
                #:G1804
                 (FUNCALL #:G1801)          ; nop
                 (FUNCALL #:G1802))))       ; *
      (FUNCALL #'#:G1803)) ; fact
    ...
```

So we wrote this program with forth but it is now lisp. We used the forth immediate words **if** and **then** to compile a conditional control structure that controls whether or not a recursion takes place. In place of a return stack, lisp will implement this recursion for us using its usual function calling infrastructure.

When testing this with **eval**, remember that the word **fact** assumes a value on the stack but we are starting with a fresh stack. To test this word we should create a wrapper word that adds the value to the stack. For example:

```
* (go-forth my-forth
    { 5 fact print } 'fact5 name)

NIL
```

Which we can execute:

```
* (eval (forth-to-lisp my-forth 'fact5))

120
NIL
```

As mentioned, because of the discrepancies between lisp and forth, our **forth-to-lisp** compiler has certain limitations. For instance, we no longer provide access to the return stack so any forth words that make use of **r>** or **>r** are forbidden. This includes **exit**, so our earlier word **countdown** will not function. However, because it doesn't use **exit**, **countdown-for-teh-hax0rz** will work. Because lisp programs are unable to access their return stacks, not all of forth's control structures can be implemented in Flub languages like COMMON LISP. Exercise: Add **exit** as a special case word that uses lisp blocks to return from a word.

Another restriction is that lisp code can't compile non-flub code so we can't translate forth words that use **compile**, such as **if**, **then**, **begin**, and **again**. Notice, of course, that forth threads created with these words can still be used to compile words, as above with **fact**. A final restriction is that, in forth, **branch-if** can jump to any thread, even if it was created inside a word different than we are currently executing. In lisp, we can only **go** to other locations within our **tagbody**. Forth allows non-local branches but general non-local branches cannot be done in Flubs like COMMON LISP.

Wait a second. Didn't we just avoid all these Flub restrictions when we were programming with our forth environment earlier? Yes. Macros allow us to convert programs to and from lisp. Thanks to macros, anything can be programmed in lisp.

Appendix A

Languages to Learn

A.1 Road to Lisp

This section contains my obviously biased thoughts on which languages I think professional lisp programmers should look into. Of course the best reason to learn any non-lisp language is that it will give you insights for use in your lisp programs.

If you are so lucky as to have learned lisp before all other programming languages, congratulations, you have made it there faster than most of us. If you have only ever learned lisp, don't worry: you haven't missed much else. All other programming languages are interesting attractions, at best enjoyable pit-stops on the road of programming knowledge: the *road to lisp*. Because lisp tends to collect and integrate all other programming ideas with merit, lisp tends to have it all. But sometimes the best ideas were not first implemented in lisp. Professional macro programmers know as many languages as possible. Once you become a lisp programmer, what it means to know a language changes. While for most programmers knowing a language means having memorised a bunch of syntax, for a lisp programmer it means knowing how the language relates to lisp.

I think every programmer should investigate all different types of languages, even if just to be able to correctly explain why lisp does it best. Go ahead, find your road to lisp.

A.2 C and Perl

After lisp, C is the most important programming language to learn. C is the *lingua franca* of the programming community. If you don't know C you will not be able to understand the majority of interesting programs out there. C is, of course, a Blub language, but it is a concisely specified and well implemented Blub language that deserves your attention. C is also a useful vehicle for learning pointer scoping[1] and for getting a feel of why it is a bad idea to design a language around the capabilities and limitations of a machine. Finally, because many modern Blubs descend from C, studying C means simultaneously studying many languages.

After lisp and then C, Perl is the most important programming language to learn, not only because of its unparalleled usefulness, but also because of its philosophy. If we have to have a Blub syntax—eliminating the possibility of macros—we may as well extend it as far as possible. Let's throw in all the possible conveniences and power-user tricks we can think of. If lisp is the result of taking syntax away, Perl is the result of taking syntax all the way. Aside from the conveniences it offers, Perl also preaches a unixy version of a very lispy principle: There's More Than One Way To Do It. Languages should not be based on style, but rather on a powerful set of primitives that, once fully understood, come together to produce unexpectedly amazing code. Programming Perl gives an instinct for finding and using programming shortcuts. Average Perl programmers even understand macro concepts like *anaphora*, and why they are useful, better than do average lisp programmers. Although Perl is a Blub, it is a beautiful Blub.

A.3 Lisp Incubators

Forth is one of my favourite programming languages. Where C took the approach of designing a language around a lowest-common-denominator computer architecture, forth took the approach of designing an ideal *abstract machine*, along with a pow-

[1] Though forth is as good or better.

erful meta-programming system that programmers are free to creatively implement around real machines. Forth is capable of extension in ways only exceeded by lisp. Chuck Moore, the inventor of forth, was actually a student of John McCarthy, inventor of lisp[EVOLUTION-FORTH-HOPL]. Forth leads to lisp.

Smalltalk is an interesting look into how object systems are supposed to work. It is worth investigating for its conceptual importance and rich history. Smalltalk can also serve as a good *gentle introduction* to modern object systems like CLOS, the COMMON LISP Object System, and even as a good introduction on how to do programming languages *right*. Smalltalk is an amusing and illuminating stop on our road to lisp. Alan Kay, the inventor of Smalltalk, tells us that lisp was one of his major inspirations. Smalltalk leads to lisp.

Haskell is an exciting, innovative language that is shaping how people think about programming. Not learning Haskell means depriving yourself of some of the most interesting research in Blub languages today. But the most important reason to gain experience with Haskell is to educate yourself on just how little static type systems actually contribute to productive programming—even ones as unobtrusive and extensible as Haskell's. Certainly, if you're proving theorems on monadic types or researching category type theory, fancy type systems are essential. But for the vast majority of programming, such type systems look a lot better *on paper* than they actually are. Haskell is an enormous success. It has proven that static typing is a failure. Ignoring static typing and infix syntax, Haskell is worth learning for its many innovative features: isolation of side-effects, lazy evaluation through graph reduction, transactional memory, etc.

Forth, Smalltalk, Haskell, and many other languages can be thought of as *lisp incubators*. Lisp incubators experiment with programming ideas, some of which become ripe enough for *pilfering* by lisp macros. All roads lead to lisp.

Appendix B

Languages to Avoid

B.1 Opinion

> There are only two industries that refer to their customers as "users".
>
> —Edward Tufte

Everything in this appendix, indeed this entire book, must be considered my *opinion*. However, I don't personally consider this appendix an opinion, but rather a brief investigation into some non-lisp language philosophies and their problems from a lisp perspective. I am not implying that writing good code is impossible in any of these languages, nor that there aren't smart people who program in and work on these languages. Please bear with me and try not to get offended. For almost all values of you, this is not about you in particular.

B.2 Blub Central

There have been Blub languages even longer than there has been lisp. Before lisp languages were standardised and decent lisp compilers written, there was often genuine need for easy to implement but poor for serious programming Blub languages. But now that our software development capabilities are so vast, why is it that

the majority of programmers use a small number of languages that are, for all intents and purposes, indistinguishable from one another? I think this can be explained with *game theory.*

Just like auto dealerships, coffee houses, and universities, often there is an observable clustering effect where establishments are built seemingly irrationally close together. If all of these establishments were to evenly distribute themselves instead of clustering, it is likely they could serve a larger total market and thus each gain larger individual profits. But because the establishments are unable to cooperate, and because if any one establishment moved to a distant location it would be worse off than if it didn't, nobody moves and the cluster remains.

There is a similar phenomenon in programming languages—one I call *Blub Central.* Most major programming languages are unable or unwilling to offer new features, or even improve their basic designs, because they are too busy trying to be like the other major programming languages. If the designers and developers of programming languages could cooperate, the result would be a much more varied, featureful family of languages, and your chances of finding a language exactly suited to your problem would increase[1]. Of course most programming language vendors are either large companies or academic committees that rarely cooperate so we have our current state of affairs: most programmers are shoe-horned into Blub Central. Because almost all programmers learn these Blub Central languages, for any one language to move from Blub Central means losing market-share.

As odd as it sounds to lisp programmers, there seems to be plausible economic explanations for Blub Central languages. The world's programmer resources that have been wasted by using the same crappy tool for every job is staggering and depressing. But fortunately for programmers everywhere there is a fast, non-stop, one-way trip out of Blub Central: the COMMON LISP express.

[1] Though recall with macros you can create that ideal language.

B.3 Niche Blub

There are literally thousands of programming languages and not all of them can be contending for Blub Central. There are, of course, *Niche Blubs*. Some of them even come fairly close to offering lisp style feature sets, although none have macro systems like lisp which gives them their Blub distinctions.

Many of these languages suffer from not having a trusted, stable standard, and also from misguided philosophies. Often they are designed around an elusive, poorly specified goal—the goal of good style. Nobody is ever really sure of where these languages should go or how they should change. Designers can and do decide to change the language to something deemed of better style, then everyone else has to run around changing their code to conform[2]. Languages that disrespect their users by imposing a *one true style* of programming should be avoided.

In lisp, there is almost always an objective *right* way to go: if it makes the language more powerful, it should be added. If it makes the language less powerful, it should be removed. COMMON LISP's stable standard is not a result of having given up on the language, but rather that there have been so few recent ideas on how to make the core language better than it already is[3]. Not to say that it will always remain so, but COMMON LISP currently represents the apex of our knowledge about programming languages. That said, Niche Blubs of course have their places. On occasion, they have even contributed valuable ideas to lisp. But why not do yourself a favour and implement your Niche Blubs as domain specific languages on COMMON LISP?

[2] And they don't have macros to ease the pain.

[3] With lisp you don't need to update the standard to extend the rest of the language.

Appendix C

Implementations

C.1 CMUCL/SBCL

The CMUCL/SBCL system actually has a history that predates COMMON LISP[1]. SBCL is a friendly fork of CMUCL. In my opinion, CMUCL is the best lisp environment currently available—with SBCL a close second. CMUCL and SBCL both make exceptional development and production environments. Both of these implementations include the amazing Python[2] optimising machine-code compiler. After a fairly trivial declaration process, COMMON LISP code compiled with Python will have performance that rivals or surpasses compilers for all other languages, especially when used on large and complicated applications. The largest difference between CMUCL and SBCL is that SBCL forces you to use Python for everything but CMUCL offers several compilers and interpreters each with their own strengths and weaknesses.

Most examples and output in this book are from CMUCL although sometimes SBCL's disassembler output is more eye-pleasing so that was used instead. CMUCL has fewer annoying, superfluous warnings and a more comfortable REPL, but SBCL can have clearer notes and is better for certain Unicode tasks.

[1] CMUCL used to be called SPICE Lisp.

[2] The name predates Python, the scripting language.

SBCL has also made certain design decisions I disagree with, such as breaking CMUCL's top-level `setq` feature, using less obvious names for system symbols, and pointlessly copying unix's getopt(3).

CMUCL and SBCL enjoy copious libraries and vibrant online user communities. It is very strange to me to hear people claim library availability is a problem for lisp. I find it quite the opposite. COMMON LISP libraries are usually better supported and of higher quality than in other languages. The foreign function interface of CMUCL/SBCL is also excellent—more flexible, stable, and efficient than any I have encountered for any other language. Libraries have never been a serious obstacle for me when developing something in COMMON LISP.

C.2 CLISP

CLISP is another excellent COMMON LISP environment. It is much more portable than CMUCL/SBCL and can run almost anywhere a C compiler can compile to. Since CLISP doesn't compile to machine-code, but instead to a byte-code format, it is often slower than CMUCL/SBCL and sometimes avoided for deployment. But deploying onto CLISP is sometimes the smartest choice if you require portability. CLISP also has remarkably fast bignum support.

C.3 Others

ECL is a COMMON LISP implementation written in C and designed to be embedded. It has roughly a third the footprint as CLISP yet manages to be fairly complete. It might be a good deployment environment for extremely memory-constrained machines. Thanks to the GMP library, it also has good bignum performance.

GCL is a COMMON LISP implementation that uses the GNU C compiler to compile lisp to native machine code through a C intermediary. It was based on the influential Kyoto CL system[KYOTO-

CL-REPORT]. It is generally not as good as CMUCL/SBCL or CLISP but still might be worth investigating.

Armed Bear is a COMMON LISP implementation that runs on top of the Java Virtual Machine. It could be useful for integrating with existing Java applications.

Perl's Parrot virtual machine has a COMMON LISP front-end that I am keeping an eye on.

Clozure CL (formerly OpenMCL) is an open source implementation of COMMON LISP that might be OK as long as it isn't run on a proprietary operating system.

There are also many proprietary implementations of COMMON LISP. Some of them are of high quality, but all suffer from critical defects—non-technical barriers that prevent you from doing or knowing things. Considering the quality and quantity of open source lisps, today there is generally no need to lock yourself into expensive, sourceless implementations or their even more limited free-trials.

Today's excellent open source COMMON LISP implementations, in combination with great free operating systems like OpenBSD and GNU/Linux, are bringing about a golden age in computing. Today there is nothing stopping programmers except the barriers of their own intellect and creativity.

Appendix D

Lisp Editors

D.1 emacs

> Editing is a rewording activity.
>
> —Alan Perlis

There is no possible way to say it without offending someone so I'll just be blunt: I think emacs is lame. Although emacs is a very flexible and powerful editor with a rich history, I would never use it. This is, of course, my extremely minority opinion. The vast majority of lisp programmers in the world are in love with emacs and its lisp programmable architecture. There is apparently an environment called ILISP which allows you to use the REPL from the editor, colour lisp syntax, auto-complete lisp forms based on the contents of your current package, whatever, but still, I would never use it.

Emacs itself includes a full lisp environment called *elisp*. Elisp is a non-lexical lisp that people have built large and sophisticated lisp applications on top of. These applications allow you to do everything from automatically make changes to the source code you are working on to browse the web and check your email from inside the editor. The ability to execute arbitrary lisp code while editing is profoundly powerful and could change the way you code, for better or for worse, but still, I would never use it.

This is what I call the *emacs trap*. In every way I've just described it, emacs sound pretty awesome. So awesome that when smart programmers try it and find it's not so awesome (it doesn't really help their coding), they somehow miss the obvious conclusion that this might be a problem not worth solving and instead start thinking of ways to pursue the potential awesomeness. The emacs trap has caused many smart lisp programmers to waste away countless hours configuring and memorising key-chords, tuning syntax colours, and writing mostly useless elisp scripts.

Emacs causes you to think about ways to write programs to do editing for you rather than ways to write programs to write programs for you. You only need to script the process of editing when the code you are editing was redundantly written. When you are completely focused on developing your application, the actual process of editing a file should be a transparent, insignificant procedure of which the mechanics never even enter your thought process. Emacs is not like that. Emacs has extension after extension, toy after toy, gimmick after gimmick for programmers to endlessly play with, be confused by, and get trapped in. Sometimes I imagine the more advanced world that might've been had the lisp geniuses that invented emacs ignored the *non-problem* that is text editing and instead focused their full attention on real programming problems.

What of Blub-style *Integrated Development Environments*? IDEs are usually massive, over-engineered emacs clones that don't even offer lisp customisation, rendering all the knobs and dials available for you to fool with even less useful than in emacs. Compared to emacs, most IDEs are a poor implementation of a bad concept—redundant editors for redundant languages.

The problem with emacs is its backwards philosophy. It treats editing as an end rather than a means. Editing itself is not an interesting process; it is what we edit that is interesting.

D.2 vi

Vi is on the opposite end of the editor spectrum from emacs. It has a canonical set of key bindings, subsets of which are univer-

sally understood by all vi users. Once you know vi you can sit down at any unix computer and edit files with zero intellectual overhead. You are at home. Instead of uncomfortable sequences of emacs key-chords, vi has a powerful modal design that lets you transparently and efficiently perform simple and complicated text manipulations. Vi was carefully designed to minimise the number of key-strokes that need to be performed and to be comfortable to use even over high-latency network connections[1].

Considering its % command that lets you copy/paste/delete/-move lisp forms, regular expressions, and flexible motion and search commands, vi lets you edit lisp code with the lowest possible overhead. In direct contrast to emacs, the meanings behind vi keys and commands never change. After a short period of use, vi becomes an extension of the primal, motor-skill section of your brain. There is something magical about being able to queue up two, three, four, or more editing commands in your head and letting your fingers carry them out while you continue thinking about your application. Unlike an IDE where a few keystrokes barely get you into some options menu, vi keystrokes are efficient, powerful, transparent, and, most importantly, constant.

For editing lisp I exclusively use Keith Bostic's nvi, the direct successor to the vi included with 4BSD. It is faster and more responsive than other editors, doesn't distract me with useless details or colours, and has never once crashed on me. I don't want anything more from an editor. Editing is one of the few things that unix got *right* and lisp got *wrong*. Worse is better. :x

[1]Which is becoming more important as technology trends towards distributed, server-side development, sometimes through long SSH chains.

References

```
;; These are the references for the book
;; _Let_Over_Lambda_ by Doug Hoyte.
;; Everything here is recommended reading.
;;
;; Each reference starts with a symbol that uniquely
;; identifies the reference. It is followed by any of the
;; following keyword arguments:
;;   :a    Author(s) as a symbol or list of symbols
;;   :y    Year of publication
;;   :n    Name or title
;;   :url  Web URL
;;   :i    Extra information or comments

(accelerating-hindsight :a kent-pitman :y 1997
  :n "Accelerating Hindsight:
      Lisp as a Vehicle for Rapid Prototyping, Web edition"
  :url "http://www.nhplace.com/kent/PS/Hindsight.html")

(aima :a (peter-norvig stuart-russell) :y 2002
  :n "Artificial Intelligence--A Modern Approach 2nd Ed.")

(algorithm-networksort :a john-gamble
  :n "Algorithm::Networksort Perl Program"
  :url "http://search.cpan.org/dist/Algorithm-Networksort/")

(amop :a gregor-kiczales :y 1991
  :n "The Art of the Metaobject Protocol")

(ansi-cl :n "ANSI Common Lisp standard described by
            the Hyperspec arranged by Kent Pitman")
(ansi-cl-iso-compatibility
  :i "ANSI adds the lambda and define-symbol-macro macros,
      ostensibly for ISO compatibility"
  :url
  "http://lisp.org/HyperSpec/Issues/iss198-writeup.html")
(ansi-cl-lisp-symbol-redefinition
  :i "ANSI clarifies what happens if you redefine symbols
      exported from the COMMON-LISP package"
  :url
  "http://lisp.org/HyperSpec/Issues/iss214-writeup.html")

(applied-cryptography :a bruce-schneier :y 1996
  :n "Applied Cryptography 2nd Ed.")
```

```
(beating-avgs :a paul-graham :y 2001
  :n "Beating the Averages"
  :url "http://paulgraham.com/avg.html")

(c-language :a (brian-kernighan dennis-ritchie) :y 1978
  :n "The C Programming Language")

(cl-ppcre :a edi-weitz
  :n "Portable Perl-compatible regular expressions
      for Common Lisp"
  :url "http://weitz.de/cl-ppcre/")

(closures-code-generation :a (marc-feeley guy-lapalme)
  :y 1987 :n "Using closures for code generation")

(clrs-algos :a (thomas-cormen charles-leiserson
                ronalid-rivest clifford-stein)
  :y 2001 :n "Introduction to Algorithms 2nd Ed.")
(clrs-algos-p705-724
  :i "Good treatment of sorting networks")

(cltl2 :a guy-steele :y 1990
  :n "Common Lisp the Language 2nd Ed.")
(cltl2-p153 :i "Chevrolet!")
(cltl2-p260
  :i "Can't define compiler macros over CL forms")
(cltl2-p530
  :i "Steele comments that backquote forms can be difficult
      to read and describes how idioms like cadar can
      conceptually represent operations like extracting
      the arguments of a lambda list")
(cltl2-p677
  :i "Can't compile lambda forms defined in a non-null
      lexical environment")
(cltl2-p685
  :i "If a function is of type compiled-function, then all macro
      calls appearing lexically within the function have already
      been expanded and will not be expanded again when the
      function is called.")
(cltl2-p967
  :i "Steele describes the backquote ladder algorithm")

(computing-cycles :a (lockwood-morris jerald-schwarz) :y 1980
  :n "Computing cyclic list structures")
```

```
(cons-should-not-cons :a henry-baker :y 1988
  :n "CONS Should not CONS its Arguments, or,
      a Lazy Alloc is a Smart Alloc")

(critique-of-dklisp :a henry-baker :y 1992
  :n "Critique of DIN Kernel Lisp Definition Version 1.2")

(design-of-cmucl :a robert-maclachlan :y 2003
  :n "Design of CMU Common Lisp"
  :url
  "http://common-lisp.net/project/cmucl/doc/CMUCL-design.pdf")

(early-cl-votes
  :n "Collection of email messages regarding the voting
      process that lead to CL. Coordinated by Steele."
  :url "www.lisp.org/table/cl-email-history.text.gz")

(elements-of-style :a nick-levine :y 2001
  :n "Lisp Elements of Style"
  :url "http://www.nicklevine.org/declarative/lectures/")

(end :a hugues-juille :y 1994
  :n "Evolving Non-Determinism: An Inventive and Efficient
      Tool for Optimization and Discovery of Strategies")

(equal-rights :a henry-baker :y 1990
  :n "Equal Rights for Functional Objects or,
      The More Things Change, The More They Are the Same")

(evolution-forth-hopl :a chuck-moore :y 1991
  :n "History Of Programming Languages:
      Forth--The Early Years"
  :url "http://www.colorforth.com/HOPL.html"
  :i "Website has unpublished text")

(evolution-forth-hopl2 :y 1993
  :a (elizabeth-rather donald-colbum chuck-moore)
  :n "History Of Programming Languages II:
      The Evolution of Forth"
  :url "http://www.forth.com/resources/evolution/index.html")

(evolution-lisp-hopl2 :a (richard-gabriel guy-steele) :y 1993
  :n "History Of Programming Languages II:
      The Evolution of Lisp")
```

```
(first-class-extents :a (shinn-der-lee daniel-friedman)
  :y 1992 :n "First-Class Extents")

(first-class-macros-have-types :a alan-bawden :y 2000
  :n "First-class macros have types")

(foundations :a haskell-curry :y 1977
  :n "Foundations of Mathematical Logic 2nd Ed.")
(foundations-p28 :i "The U language")
(foundations-p31-footnote3
  :i "Description of Quine's quasiquotation")

(function-of-function :a joel-moses :y 1970
  :n "The Function of FUNCTION in LISP, or Why the FUNARG
      Problem Should be Called the Environment Problem")

(gc-is-fast :a andrew-appel :y 1987
  :n "Information Processing Letters: Garbage Collection
      can be faster than Stack Allocation")

(generation-scavenging :a david-ungar :y 1984
  :n "Generation Scavenging: A Non-Disruptive High
      Performance Storage Reclamation Algorithm")

(graham-ansi-cl :a paul-graham :y 1995
  :n "ANSI Common Lisp")

(graphics-gems-p171-175 :a alan-paeth :y 1990
  :n "Graphics Gems: Median Finding on a 3-by-3 Grid")

(growing-a-language :a guy-steele :y 1999
  :n "Growing a Language")

(history-of-cl :a masayuki-ida :y 2002
  :n "The History of Lisp Standardization during 1984-1990"
  :i "Interesting Japanese perspective on CL")

(history-of-lisp :a john-mccarthy :y 1979
  :n "History of Lisp"
  :url
  "http://www-formal.stanford.edu/jmc/history/lisp/lisp.html")

(hygiene-by-renaming :a william-clinger :y 1991
  :n "Hygienic macros through explicit renaming"
```

```
  :i "Interesting relation to sub-lexical scope")

(hygienic-macro-expansion :y 1986
  :a (eugene-kohlbecker daniel-friedman
      matthias-felleisen bruce-duba)
  :n "Hygienic Macro Expansion"
  :i "One of the earliest discussions of hygienic macros")

(implementations :a daniel-weinreb :y 2007
  :n "Common Lisp Implementations: A Survey"
  :url "http://common-lisp.net/~dlw/LispSurvey.html")

(interlisp :a warren-teitelman :y 1976
  :n "Interlisp Reference Manual")

(interlisp-tops20 :a warren-teitelman :y 1981
  :n "Interlisp Documentation for TOPS-20 operating system")

(kyoto-cl-report :a (taiichi-yuasa masami-hagiya) :y 1985
  :n "Kyoto Common Lisp Report")

(lambda-papers :a (guy-steele gerald-sussman)
  :n "The Lambda Papers"
  :url "http://library.readscheme.org/page1.html"
  :i "Lambda, the Ultimate topic")

(linear-lisp :a henry-baker :y 1992
  :n "Lively Linear Lisp, or Look Ma, No Garbage!")

(linear-lisp-and-forth :a henry-baker :y 1993
  :n "Linear Logic and Permutation Stacks--
      The Forth Shall Be First")

(lisp1-5 :a (john-mccarthy paul-abrahams daniel-edwards
             timothy-hart michael-levin)
  :y 1962 :n "Lisp 1.5 programmer's manual")

(lisp2-4life :a (richard-gabriel kent-pitman) :y 1988
  :n "International Journal on Lisp and Symbolic Computation:
      Technical issues of separation in function cells and
      value cells."
  :url "http://www.dreamsongs.com/Separation.html")

(lisp-style :a (peter-norvig kent-pitman) :y 1993
  :n "Tutorial on Good Lisp Programming Style")
```

```
(macaroni :a guy-steele :y 1977
  :n "Macaroni is better than spaghetti")

(maclisp :a david-moon :y 1974
  :n "Maclisp Manual, aka the Moonual")

(maclisp-revised :a kent-pitman :y 1983
  :n "The Revised Maclisp Manual, aka the Pitmanual")

(maclisp-revised-again :a kent-pitman :y 2007
  :n "The Revised Maclisp Manual, Sunday Morning Edition"
  :url "http://www.maclisp.info/pitmanual/index.html")

(macro-by-example :a (eugene-kohlbecker mitchell-wand) :y 1987
  :n "Macro-by-example: Deriving syntactic transformations
      from their specifications")

(macro-definitions :a timothy-hart :y 1963
  :n "MIT AI Memo: MACRO Definitions for LISP")

(macros-that-compose :a oleg-kiselyov :y 2002
  :n "Macros that Compose: Systematic Macro Programming")

(moving-forth :a bradford-rodriguez
  :n "The Computer Journal: Moving Forth Column"
  :url "http://www.zetetics.com/bj/papers/")

(netcat :a *hobbit* :y 1996
  :n "Netcat--The Network Swiss Army Knife")

(nil-perspective :a jonl-white :y 1979
  :n "MACSYMA conference: NIL--A Perspective")

(objects-have-failed :a richard-gabriel :y 2002
  :n "OOPSLA: Objects Have Failed"
  :url
  "http://www.dreamsongs.com/ObjectsHaveFailedNarrative.html")
(objects-have-not-failed :a guy-steele :y 2002
  :n "OOPSLA: Objects Have Not Failed"
  :url
  "http://www.dreamsongs.com/ObjectsHaveNotFailedNarr.html")

(oleg-website :a oleg-kiselyov
  :url "http://okmij.org/ftp/")
```

```
(on-lisp :a paul-graham :y 1993 :n "On Lisp")
(on-lisp-p25
  :i "Lambda forms returned from a compiled function will
      themselves also be compiled")
(on-lisp-p108
  :i "Graham doesn't like using the calling environment
      with macros")
(on-lisp-p176 :i "sortf")
(on-lisp-p223 :i "defanaph")
(on-lisp-p237 :i "with-places")

(opening-closures :a (jeffrey-siskind barak-pearlmutter)
  :y 2007 :n "First-Class Nonstandard Interpretations
              by Opening Closures")

(paip :a peter-norvig :y 1992
  :n "Paradigms of Artificial Intelligence Programming")
(paip-pix :i "Pascal/C are special purpose, lisp is general")
(paip-p20 :i "Why do we call it lambda?")
(paip-p323
  :i "Another way to avoid destructuring overhead")
(paip-p783 :i "Answer 22.10: Norvig says that he would have
               gladly used the lambda macro")
(paip-p853
  :i "Once-Only: A lesson in Macrology")
(paip-p877
  :i "Don't mix &optional and &key parameters")

(patterns-of-software :a richard-gabriel :y 1998
  :n "Patterns of Software: Tales from the Software Community")

(performance-evaluation :a richard-gabriel :y 1985
  :n "Performance and Evaluation of Lisp Systems")

(pointers-as-closures :a oleg-kiselyov
  :n "Comp.lang.scheme: How to emulate & in Scheme"
  :url "http://okmij.org/ftp/Scheme/pointer-as-closure.txt")

(practical-cl :a peter-seibel :y 2005
  :n "Practical Common Lisp")
(practical-cl-p95 :i "Interesting loopy once-only")

(pragmatic-parsing :a henry-baker :y 1991
  :n "Pragmatic Parsing in Common Lisp; or,
```

```
      Putting defmacro on Steroids"
  :i "Lots of great C language digs and a challenge to Scheme")
(pragmatic-parsing-p11-footnote20
  :i "Mixing of &optional and &keys bites even the experts")

(programming-perl :a (larry-wall tom-christiansen jon-orwant)
  :y 2000 :n "Programming Perl 3rd Ed.")

(quasiquotation :a alan-bawden :y 1999
  :n "Quasiquotation in Lisp"
  :i "Excellent, comprehensive history")

(r5rs :a (richard-kelsey william-clinger jonathan-rees)
  :n "Revised 5 Report on the Algorithmic Language Scheme")
(r5rs-p25
  :i "Scheme booleans are dumb")

(schematics :a (vincent-manis james-little) :y 1995
  :n "Schematics of Computation")
(schematics-p301-343 :i "Let Over Lambda")
(schematics-p466-469 :i "Tlists")

(scheme-revisited :a (guy-steele gerald-sussman) :y 1998
  :n "HOSC: The First Report on Scheme Revisited")

(sicp :a (harold-abelson gerald-sussman julie-sussman)
  :y 1985
  :n "Structure and Interpretation of Computer Programs")

(small-pieces :a christian-queinnec :y 1994
  :n "Lisp in Small Pieces")
(small-pieces-p46
  :i "IS-Lisp and EuLisp have enforced separate namespaces
      for dynamic/lexical variables")
(small-pieces-p340-341
  :i "Code-walking is hard in Scheme")

(sn-applications :a ken-batcher :y 1968
  :n "Sorting networks and their applications")

(spaghetti-stacks :a (daniel-bobrow ben-wegbreit) :y 1973
  :n "A model and stack implementation of multiple environments")

(special-forms :a kent-pitman :y 1980
  :n "Special Forms in Lisp")
```

```
(spice-internals :a (scott-fahlman guy-steele
                     gail-kaiser walter-roggen) :y 1981
  :n "Internal Design of Spice Lisp")

(starting-forth :a leo-brodie :y 1981
  :n "Starting Forth"
  :i "One of the best tutorials for learning forth")

(syntactic-abstraction :y 1993
  :a (kent-dybvig robert-hieb carl-bruggeman)
  :n "Syntactic Abstraction in Scheme"
  :i "Describes the hygiene system used by most Schemes")

(syntactic-closures :a (alan-bawden jonathan-rees) :y 1988
  :n "Syntactic Closures")

(syntactically-recursive :a christian-queinnec :y 1993
  :n "Compiling Syntactically Recursive Programs")

(syntax-rules-insane :a al-petrofsky :y 2002
  :n "Comp.lang.scheme: An Advanced Syntax-Rules Primer
      for the Mildly Insane")

(syntax-rules-unhygienic :a oleg-kiselyov :y 2002
  :n "How to Write Seemingly Unhygienic and Referentially
      Opaque Macros with Syntax-rules")

(taocp-vol3 :a donald-knuth :y 1973
  :n "The Art of Computer Programming Volume 3:
      Sorting and Searching 2nd Ed.")
(taocp-vol3-p111
  :i "Algorithm 5.2.2M: Merge Exchange Sort")

(thinking-forth :a leo-brodie :y 1984
  :n "Thinking Forth: A Language and Philosophy
      for Solving Problems")

(threading-lisp :a david-nordstrom :y 1990
  :n "Threading Lisp")

(unified-theory-of-gc :a (david-bacon perry-cheng v-t-rajan)
  :y 2004 :n "A Unified Theory of Garbage Collection")

(useful-lisp-algos1-chapter3 :a (richard-waters peter-norvig)
```

```
  :y 1993 :n "Some Useful Lisp Algorithms: Part 1, Chapter 3,
              Implementing Queues in Lisp")

(useful-lisp-algos2 :a richard-waters :y 1993
  :n "Some Useful Lisp Algorithms: Part 2"
  :i "This report has 3 chapters, all are great:
      1: pretty printing
      2: useful code-walking macro: macroexpand-all
      3: list accumulation performance")

(winston-horn :a (patrick-winston berthold-horn) :y 1988
  :n "Lisp 3rd Ed.")

(worse-is-better :a richard-gabriel :y 1991
  :n "Lisp: Good News, Bad News, How to Win Big, or
      Worse Is Better"
  :url "http://www.dreamsongs.com/WIB.html")

(xml-as-read-macro :a sam-steingold
  :n "CLOCC source code: XML implemented as a read macro")
```

References by Author

HOBBIT (NETCAT)

AL-PETROFSKY (SYNTAX-RULES-INSANE)

ALAN-BAWDEN (SYNTACTIC-CLOSURES QUASIQUOTATION
FIRST-CLASS-MACROS-HAVE-TYPES)

ALAN-PAETH (GRAPHICS-GEMS-P171-175)

ANDREW-APPEL (GC-IS-FAST)

BARAK-PEARLMUTTER (OPENING-CLOSURES)

BEN-WEGBREIT (SPAGHETTI-STACKS)

BERTHOLD-HORN (WINSTON-HORN)

BRADFORD-RODRIGUEZ (MOVING-FORTH)

BRIAN-KERNIGHAN (C-LANGUAGE)

BRUCE-DUBA (HYGIENIC-MACRO-EXPANSION)

BRUCE-SCHNEIER (APPLIED-CRYPTOGRAPHY)

CARL-BRUGGEMAN (SYNTACTIC-ABSTRACTION)

CHARLES-LEISERSON (CLRS-ALGOS)

CHRISTIAN-QUEINNEC (SYNTACTICALLY-RECURSIVE SMALL-PIECES)

CHUCK-MOORE (EVOLUTION-FORTH-HOPL2 EVOLUTION-FORTH-HOPL)

CLIFFORD-STEIN (CLRS-ALGOS)

DANIEL-BOBROW (SPAGHETTI-STACKS)

DANIEL-EDWARDS (LISP1-5)

DANIEL-FRIEDMAN (HYGIENIC-MACRO-EXPANSION FIRST-CLASS-EXTENTS)

DANIEL-WEINREB (IMPLEMENTATIONS)

DAVID-BACON (UNIFIED-THEORY-OF-GC)

DAVID-MOON (MACLISP)

DAVID-NORDSTROM (THREADING-LISP)

DAVID-UNGAR (GENERATION-SCAVENGING)

DENNIS-RITCHIE (C-LANGUAGE)

DONALD-COLBUM (EVOLUTION-FORTH-HOPL2)

DONALD-KNUTH (TAOCP-VOL3)

EDI-WEITZ (CL-PPCRE)

ELIZABETH-RATHER (EVOLUTION-FORTH-HOPL2)

EUGENE-KOHLBECKER (MACRO-BY-EXAMPLE HYGIENIC-MACRO-EXPANSION)

GAIL-KAISER (SPICE-INTERNALS)

GERALD-SUSSMAN (SICP SCHEME-REVISITED LAMBDA-PAPERS)

GREGOR-KICZALES (AMOP)

GUY-LAPALME (CLOSURES-CODE-GENERATION)

GUY-STEELE (SPICE-INTERNALS SCHEME-REVISITED
OBJECTS-HAVE-NOT-FAILED MACARONI LAMBDA-PAPERS
GROWING-A-LANGUAGE EVOLUTION-LISP-HOPL2 CLTL2)

HAROLD-ABELSON (SICP)

HASKELL-CURRY (FOUNDATIONS)

HENRY-BAKER (PRAGMATIC-PARSING LINEAR-LISP-AND-FORTH
LINEAR-LISP EQUAL-RIGHTS CRITIQUE-OF-DKLISP
CONS-SHOULD-NOT-CONS)

HUGUES-JUILLE (END)

JAMES-LITTLE (SCHEMATICS)

JEFFREY-SISKIND (OPENING-CLOSURES)

JERALD-SCHWARZ (COMPUTING-CYCLES)

JOEL-MOSES (FUNCTION-OF-FUNCTION)

JOHN-GAMBLE (ALGORITHM-NETWORKSORT)

JOHN-MCCARTHY (LISP1-5 HISTORY-OF-LISP)

JON-ORWANT (PROGRAMMING-PERL)

JONATHAN-REES (SYNTACTIC-CLOSURES R5RS)

JONL-WHITE (NIL-PERSPECTIVE)

JULIE-SUSSMAN (SICP)

KEN-BATCHER (SN-APPLICATIONS)

KENT-DYBVIG (SYNTACTIC-ABSTRACTION)

KENT-PITMAN (SPECIAL-FORMS MACLISP-REVISED-AGAIN
MACLISP-REVISED LISP-STYLE LISP2-4LIFE
ACCELERATING-HINDSIGHT)

LARRY-WALL (PROGRAMMING-PERL)

LEO-BRODIE (THINKING-FORTH STARTING-FORTH)

LOCKWOOD-MORRIS (COMPUTING-CYCLES)

MARC-FEELEY (CLOSURES-CODE-GENERATION)

MASAMI-HAGIYA (KYOTO-CL-REPORT)

MASAYUKI-IDA (HISTORY-OF-CL)

MATTHIAS-FELLEISEN (HYGIENIC-MACRO-EXPANSION)

MICHAEL-LEVIN (LISP1-5)

MITCHELL-WAND (MACRO-BY-EXAMPLE)

NICK-LEVINE (ELEMENTS-OF-STYLE)

```
OLEG-KISELYOV (SYNTAX-RULES-UNHYGIENIC POINTERS-AS-CLOSURES
 OLEG-WEBSITE MACROS-THAT-COMPOSE)

PATRICK-WINSTON (WINSTON-HORN)

PAUL-ABRAHAMS (LISP1-5)

PAUL-GRAHAM (ON-LISP GRAHAM-ANSI-CL BEATING-AVGS)

PERRY-CHENG (UNIFIED-THEORY-OF-GC)

PETER-NORVIG (USEFUL-LISP-ALGOS1-CHAPTER3 PAIP LISP-STYLE AIMA)

PETER-SEIBEL (PRACTICAL-CL)

RICHARD-GABRIEL (WORSE-IS-BETTER PERFORMANCE-EVALUATION
 PATTERNS-OF-SOFTWARE OBJECTS-HAVE-FAILED
 LISP2-4LIFE EVOLUTION-LISP-HOPL2)

RICHARD-KELSEY (R5RS)

RICHARD-WATERS (USEFUL-LISP-ALGOS2 USEFUL-LISP-ALGOS1-CHAPTER3)

ROBERT-HIEB (SYNTACTIC-ABSTRACTION)

ROBERT-MACLACHLAN (DESIGN-OF-CMUCL)

RONALID-RIVEST (CLRS-ALGOS)

SAM-STEINGOLD (XML-AS-READ-MACRO)

SCOTT-FAHLMAN (SPICE-INTERNALS)

SHINN-DER-LEE (FIRST-CLASS-EXTENTS)

STUART-RUSSELL (AIMA)

TAIICHI-YUASA (KYOTO-CL-REPORT)

THOMAS-CORMEN (CLRS-ALGOS)

TIMOTHY-HART (MACRO-DEFINITIONS LISP1-5)

TOM-CHRISTIANSEN (PROGRAMMING-PERL)
```

V-T-RAJAN (UNIFIED-THEORY-OF-GC)

VINCENT-MANIS (SCHEMATICS)

WALTER-ROGGEN (SPICE-INTERNALS)

WARREN-TEITELMAN (INTERLISP-TOPS20 INTERLISP)

WILLIAM-CLINGER (R5RS HYGIENE-BY-RENAMING)

Index

abstract
    machines, 289, 297, 342
    registers, 286, 297
alet over alambda, 162
alet over dlambda, 174
ambivalent, 73
amortisation, 238, 241, 251
anaphora, 126, 153, 181, 342
    block, 173
    chaining, 203
    closing of, 174, 177
    injection of, 166
    read, 156
anaphoric macro, 249
annoying, 230
anonymous
    classes, 34
    functions, 151
arity, 151
artificial intelligence, 263
attack vectors, 105
automatic
    anaphora, 153
    gensyms, 60
    once-only, 67

backquote, 78
    nested, 61, 114
Baker, Henry, 285
base case, 136
Batcher, Ken, 261
benchmarking, *see* bullshitting
big deal, 1
big picture, 226
Big-O, 277
Blub, 10
    Central, 346
    Niche, 347
booleans
    explicit, 155
    generalised, 154
boot-strapping, 12
Bostic, Keith, 355
boundary values, 150
brevity, 296
    theory of, 296, 297
buffer overflows, 238
bullshitting, 224, 277

C, 342
cells
    cons, 20
    memory, 290
chaining, 113
Chevrolet, 126
CL-PPCRE, 87
class variables, 36
CLOS, 18, 174, 187
closure, *see* let over lambda
code-walking, 15, 64, 120, 126, 166
    mental, 83
cohesion, 88
compiled expansions, 225
compiler, 269
    notes, 230
    sufficiently smart, 272
conditional probability, 257
cons pool, 248
constant folding, 40, 214
constant time, 245
contamination, 265
control stack, 324

counter, 32
crossing over, 52
Curry, Haskell, 7
cycle, 81, 97

declarations, 19, 210
denial of service, 102
dereference, 234
destructuring, 148, 149
    at run-time, 148, 176
    defmacro, 105, 217
    lambda, 149, 217
dictionary, 288
Dijkstra, Edsger, 39
directed acyclic graphs, 95, 318
dispatching, 149
divide and conquer, 147
domain specific languages, 40
duality of syntax, 71, 94, 158, 182, 188, 192, 202, 209, 216, 249, 296
    theory of, 303
dynamic closures, *see* spaghetti stack

earmuffs, 22, 303
eating characters, 95
elisp, 353
emacs
    trap, 354
error checking, 116
escaping characters, 84
evolutionary algorithms, 263
Evolving Non-Determinism, 263
exporting closed bindings, 200

fail-safe, 167
first-class, 194
    procedures, 109
floating point vs fixed point, 226
Flub, 291, 319
    flubifying, 332
    pure, 320
forth, 241, 285, 342
    words, 288
FORTRAN in any language, 4
free variable, 32, 48
    injection, 50, 171, 178
    U-language, 153
friction, 303
full recursion, 330
funcallable instances, 108
function composition, 122
function designator, *see* lambda
functional style, 108

G-bang symbol, 59
Gabriel, Richard, 56
Gamble, John, 254
game theory, 346
garbage collectors, 248
generalised
    code, 209
    variables, 189, 192
generalised boolean, 306
generated symbols, *see* gensyms
generational collection, 251
gensyms, 54, 233
gentle introduction, 343
getopt, 350
goto, 131, 322
Graham, Paul, 4, 10, 13, 29, 50, 153, 207, 254

hacking, *see* r00ting
Hart, Timothy, 2
hash-table, 97
Haskell, 110, 343
high-level, 132
Hoare, C.A.R., 111
holes, 298, 304
hooks, 75
hosed, 243
hot-spots, 238
hotpatching, 175, 196
Hoyte, Brian, 285

if, 159
    numeric, 51
immediacy, 289, 304, 305, 323
impedance mismatch, 98
implicit

bindings, 49
contexts, 120
lambdas, 34, 120
lexical variables, 126
progns, 27, 120
implicitisation, 183
in-place, 277
incremental collection, 251
incubators, 343
indefinite extent, 18
indirection, 162, 164, 168, 228
inheritance, 37, 67
inline, 137, 229, 245, 291
threshold, 137
inner interpreter, 290
input, 99
Integrated Development Environments, 354
inter-closure protocol, 192
Interlisp, 47, 244
internable, 209
interned, 58
iterative development, 10
iteratively, 297

java, 351
jihad, 35

Kay, Alan, 126
keyword symbols, 147
Kiselyov, Oleg, 233
kludge, 155, 194
Knuth, Donald, 261

ladder of quotation, 114
lambda, 27, 159
folding, 30
form, 28
lambda over let over lambda, 33
let over dlambda, 149
let over lambda, 17, 31, 72, 156, 194, 200
let over two lambdas, 33
lexical
invisibility, 167
transparency, 126, 155
lexical closures, *see* let over lambda
lingua franca, 342
Lisp in Small Pieces, 187
lisp machine, 65, 286
lisp printer, 100
lisp reader, 100
lisp-1, 55
lisp-2, 55
locality, 209, 248
loop unrolling, 261
low-level, 132

Maclisp, 135
macroexpand, 46
macroexpand-all, 225
macros
age of, 3
anaphoric, 153
combinations of, 6, 50, 139, 153, 176, 202, 251, 328
compiler, 210, 216
hygienic, 54, 56
inside out, 164, 174, 190
local, 131
pandoric, 189
read, 59, 75, 210
symbol, 189, 194
unhygienic, 155
math, 107
McCarthy, John, 1
McMahon, Mike, 83
median, 265
MEROONET, 187
messages, 147
meta-programming, 10
methods, 147, 151, 202
Monty Hall problem, 259
Moore, Chuck, 286

naked primitive, 320
named let, 45
natural residue, 326
nesting, 105, 215, 325
netcat, 101
Nmap, 99

non-problem, 83, 354
normal balanced brackets, 313
Norvig, Peter, 64, 221
null lexical environment, 23, 204
nvi, 355

O-bang symbols, 66
object-oriented, 187
objects, 32
obliviously, 254
Ocaml, 110
On Lisp, 4
on paper, 343
one true style, 347
open closures, 189
opinion, 345

packages, 54
Paeth, Alan, 254
PAIP, 64, 221
Pandora's box, 187
performance myth, 87, 207
Perl, 85, 87, 342
Perlis, Alan, 75, 353
pilfering, 85, 90, 343
pipeline, 227
Pitman, Kent, 56
plug-ins, 121
pointer to a pointer, 234
portable, 87
pretty printing, 82
primitives, 297
    meta, 289
procedures, 108
program
    counter, 132, 288
    listings, 8
    state, 32, 166
pseudo-code, 208

quasiquote, *see* backquote
Queinnec, Christian, 187
queues, 244
Quine, Willard, 83
quines, 83
quote, 69

r00ting, 98
readtable, 102
recursion
    anonymous, 331
    double, 334
    named, 331
    of macros, 135
    tail, 130, 321
referential transparency, 49, 53, 71
regular expressions, 87
REPL, 9
restarting, 99
return address, 287
right, 56, 99, 155, 195, 221, 227, 231, 343, 347, 355
road to lisp, 341
rounding, 113
run-away lane, 241
Russell, Steve, 21

S-expressions, 88
safe zones, 241
safe-mode, 230
schema, 105
Schematics of Computation, 244
Scheme, 243
scope, 20
    dynamic, 22
    lexical, 20
    pointer, 21, 232
    sub-lexical, 178
    super sub-lexical, 184
secret weapon, 7
security, *see* r00ting
segfault, 242
Seibel, Peter, 99
selection networks, 255
self-referential, 59, 320
serialising, 97
shadowing, 23, 69, 215
shell code, 101
SICP, 243
side-effects, 52, 65, 81
slang, 22
Smalltalk, 343
sort algorithm

merge, 261
merge exchange, 261
quick, 254
shell, 261
spaghetti stack, 73
stack, 19
stack frame, 287
standard library, 299
static variables, 36
Steele, Guy, 17, 78, 84, 114, 126
stop, 60, 112
structured programming, 131
structures, 288
substitution, 90
symbol, 28

tagging, 26
Teitelman, Warren, 244
temporary extent, 19
theory, 268
threaded code, 289, 297
cons, 293
direct, 290
indirect, 290
subroutine, 291
token, 291
tlist, 244
top-down, 111
transparent specification, 133, 153, 246, 311, 325
trees, 95
shaking, 319
Tufte, Edward, 345
tunnel, 205, 236
type inference, 231

ugly printed, 82
unbound, 23
Unicode, 349
unquote, 61, 78
use cases, 111, 298
utilities, 13

validation, 105
variable capture, 48, 49
at run-time, 233
vi, 354

weird, 11, 286
Weitz, Edi, 87, 225
worst-case, 259
wrapper, 247
write more to get less, 209
wrong, 56, 227, 355

XML, 76

"But I want scrambled!"
Image from Thinking Forth by Leo Brodie.
Used with permission of the author.

Let Over Lambda is one of the most hardcore computer programming books out there. Starting with the fundamentals, it describes the most advanced features of the most advanced language: COMMON LISP. Only the top percentile of programmers use lisp and if you can understand this book you are in the top percentile of lisp programmers.

This book is about macros, that is programs that write programs. Macros are what make lisp the greatest programming language in the world. When used properly, macros enable amazing feats of abstraction, programmer productivity, and code efficiency and security that are unheard of elsewhere. Macros let you do things you simply cannot do in other languages.

If you are looking for a dry coding manual that re-hashes common-sense techniques in whatever langue du jour, this book is not for you. This book is about pushing the boundaries of what we know about programming. While this book teaches useful macro skills that can help solve your programming problems today and now, it has also been designed to be entertaining and inspiring. If you have ever wondered what lisp or even programming itself is really about, this is the book you have been looking for.

```
(defmacro! sortf (comparator &rest places)
  (if places
      `(tagbody
        ,@(mapcar
            #`(let ((,g!a #1=,(nth (car a1) places))
                    (,g!b #2=,(nth (cadr a1) places)))
                (if (,comparator ,g!b ,g!a)
                    (sotf #1# ,g!b
                          #2# ,g!a)))
            (build-batcher-sn (length places))))))
```

λ

ISBN 978-1-4357-1275-1
90000
9 781435 712751